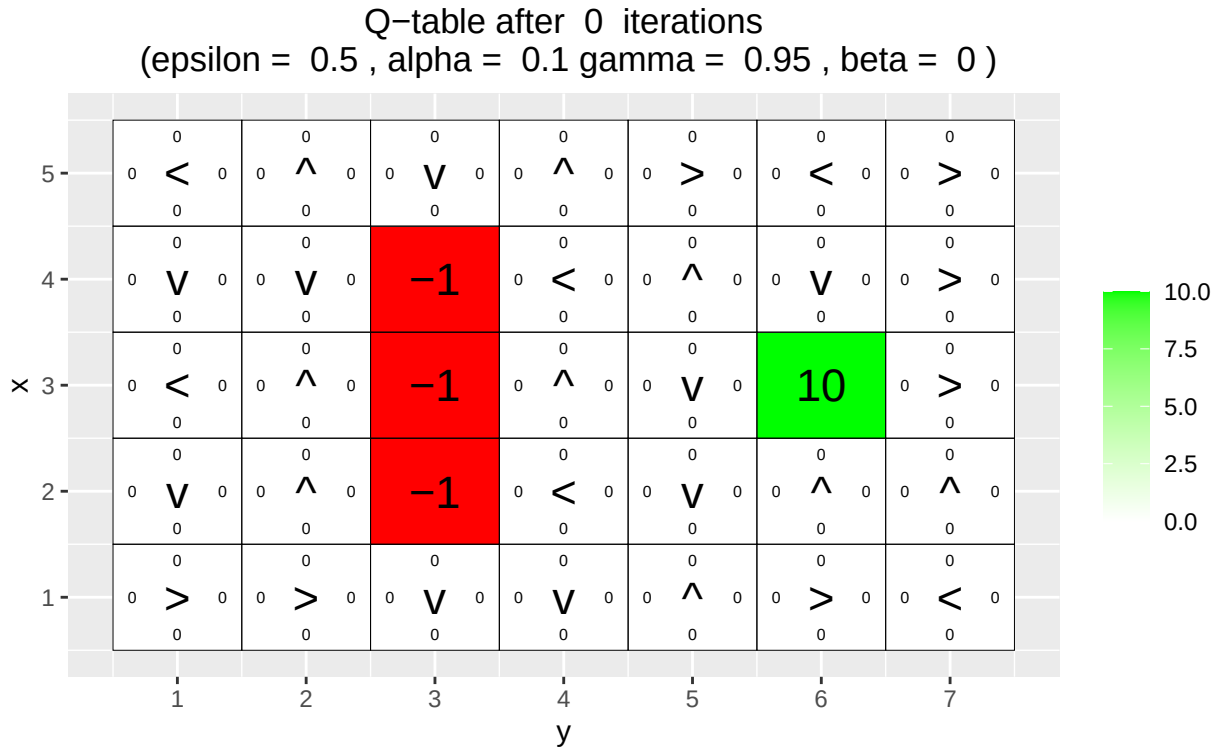
LAB 3: REINFORCEMENT LEARNING

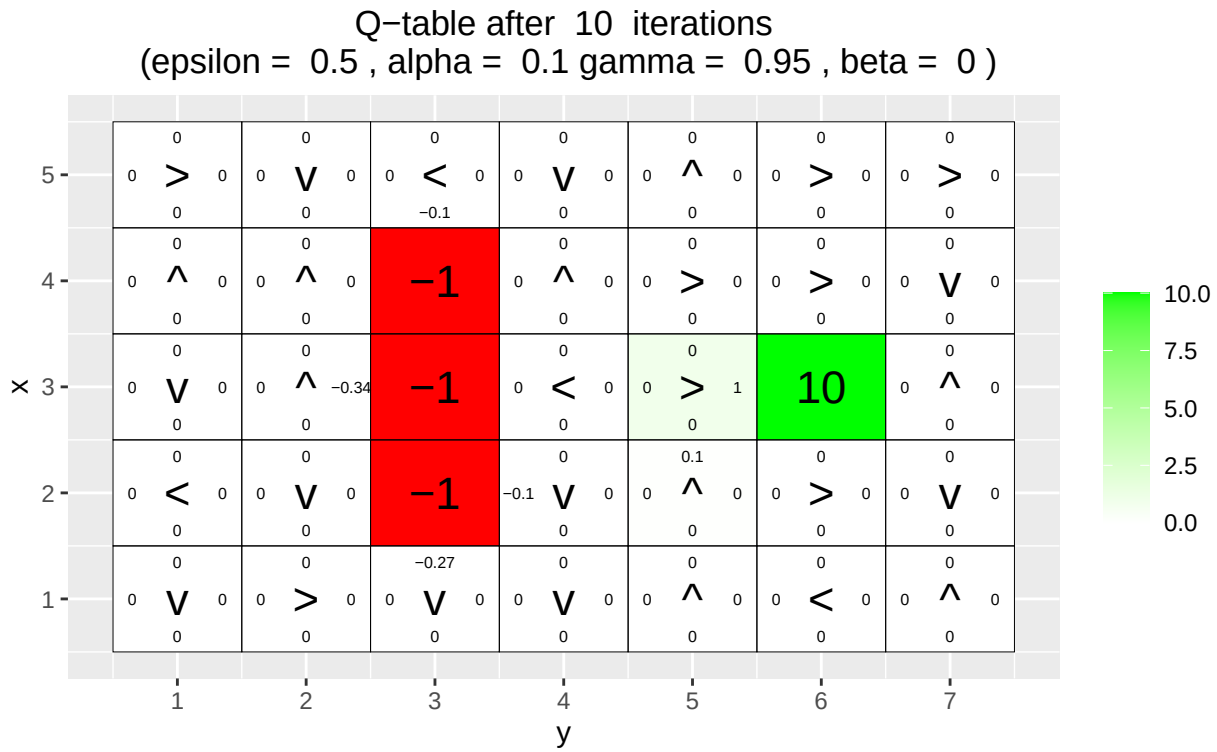
Namita Sharma

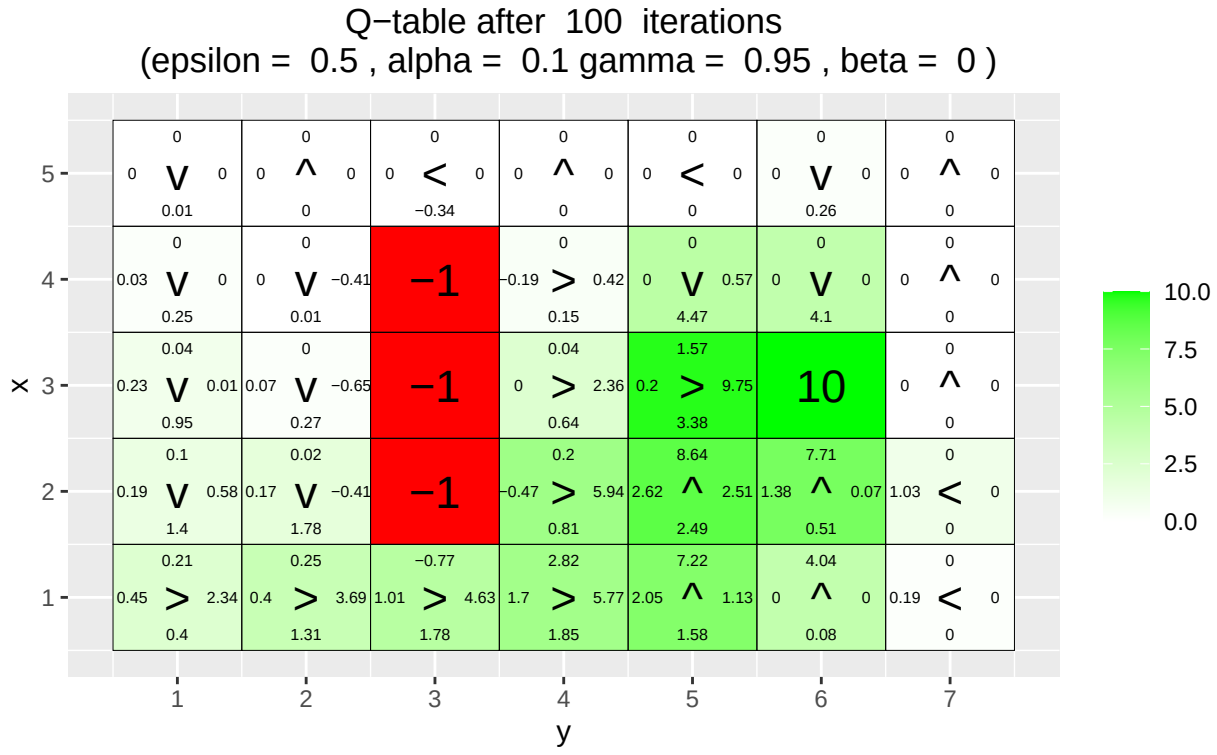
10/6/2020

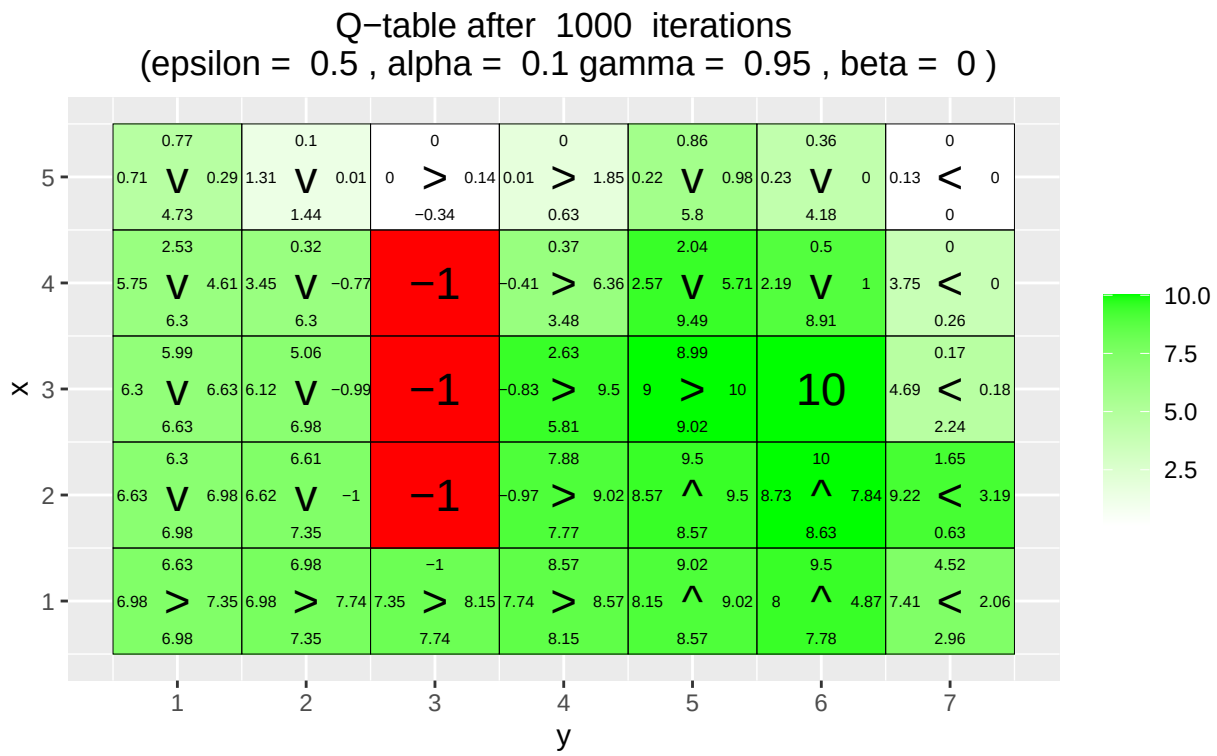
Q-Learning

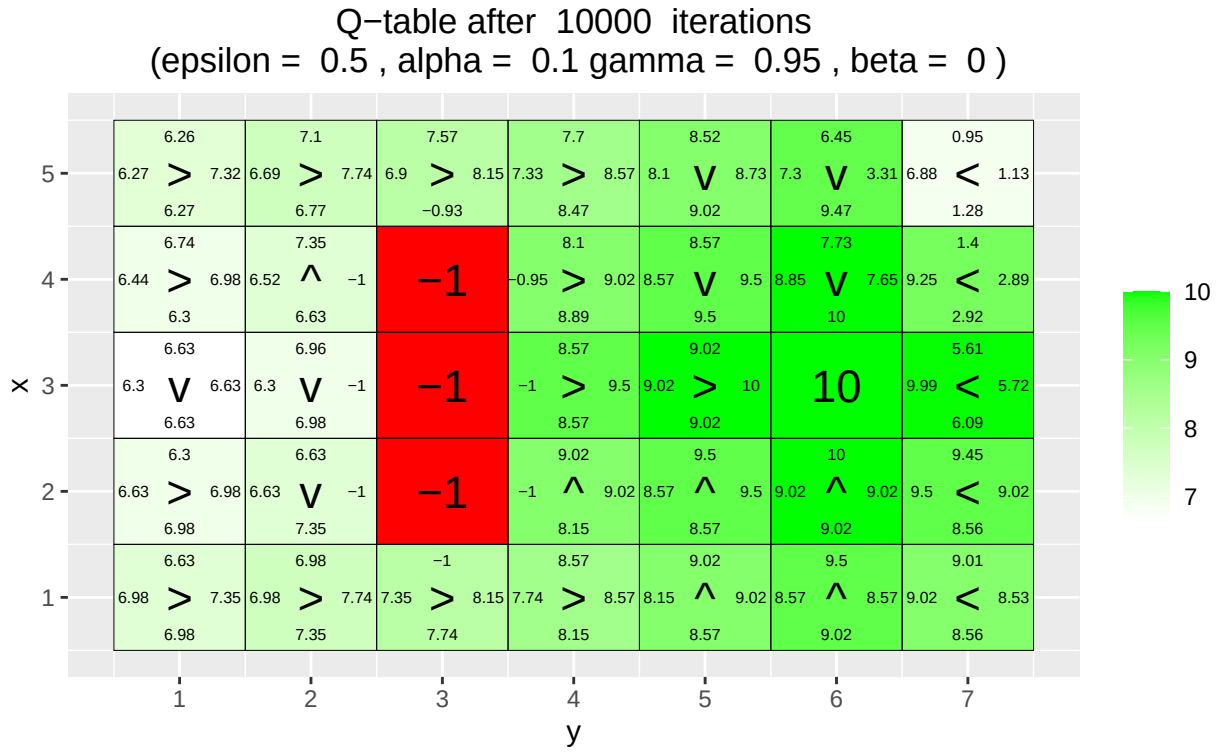
Environment A






Q−table after 100 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0


Q−table after 1000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10.0
7.5
5.0
2.5



- What has the agent learned after the first 10 episodes ?

The agent has not learnt much in the first 10 episodes except to maybe avoid the terminal states with negative rewards and explore the search space around it instead. Naturally, the states closer to the initial state of the agent were visited more often than the states farther away and this can be seen in the action values updated in the Q-table for the states leading up to these terminal states. The agent has not had enough time to explore the environment beyond these terminal states yet.

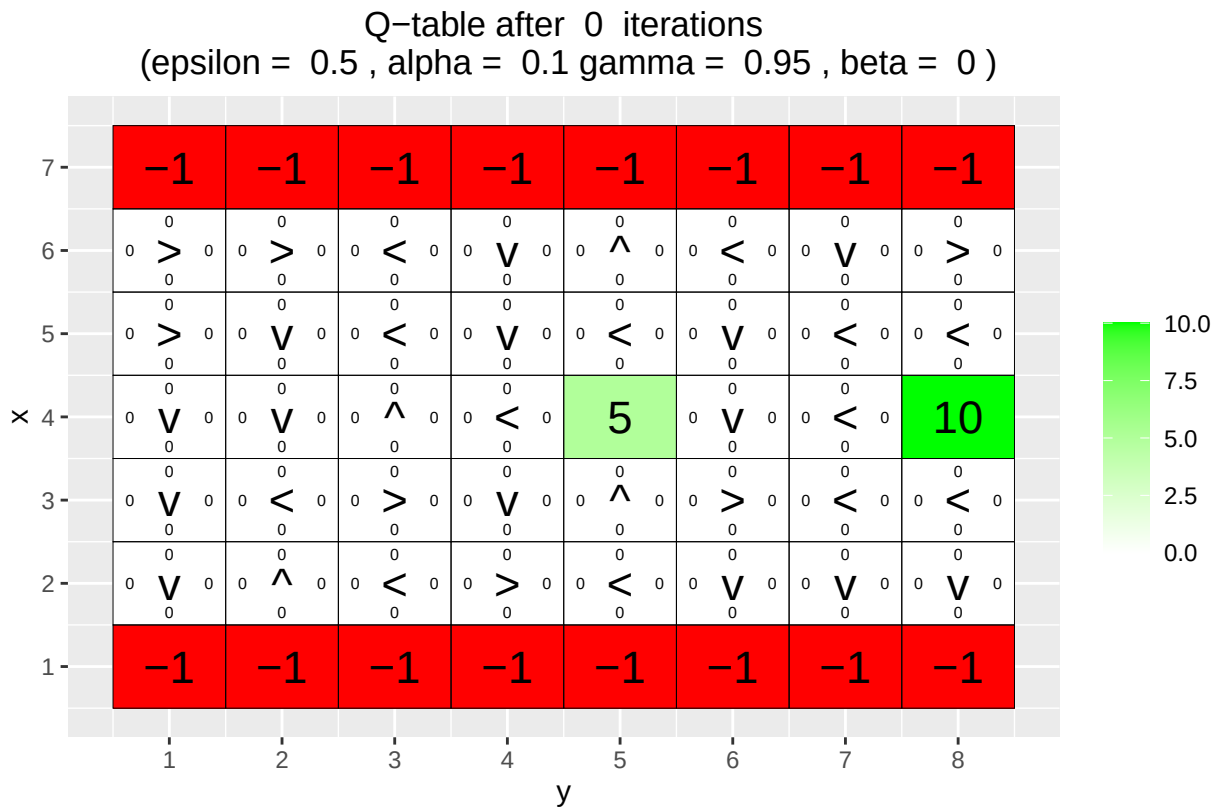
- Is the final greedy policy (after 10000 episodes) optimal? Why / Why not ?

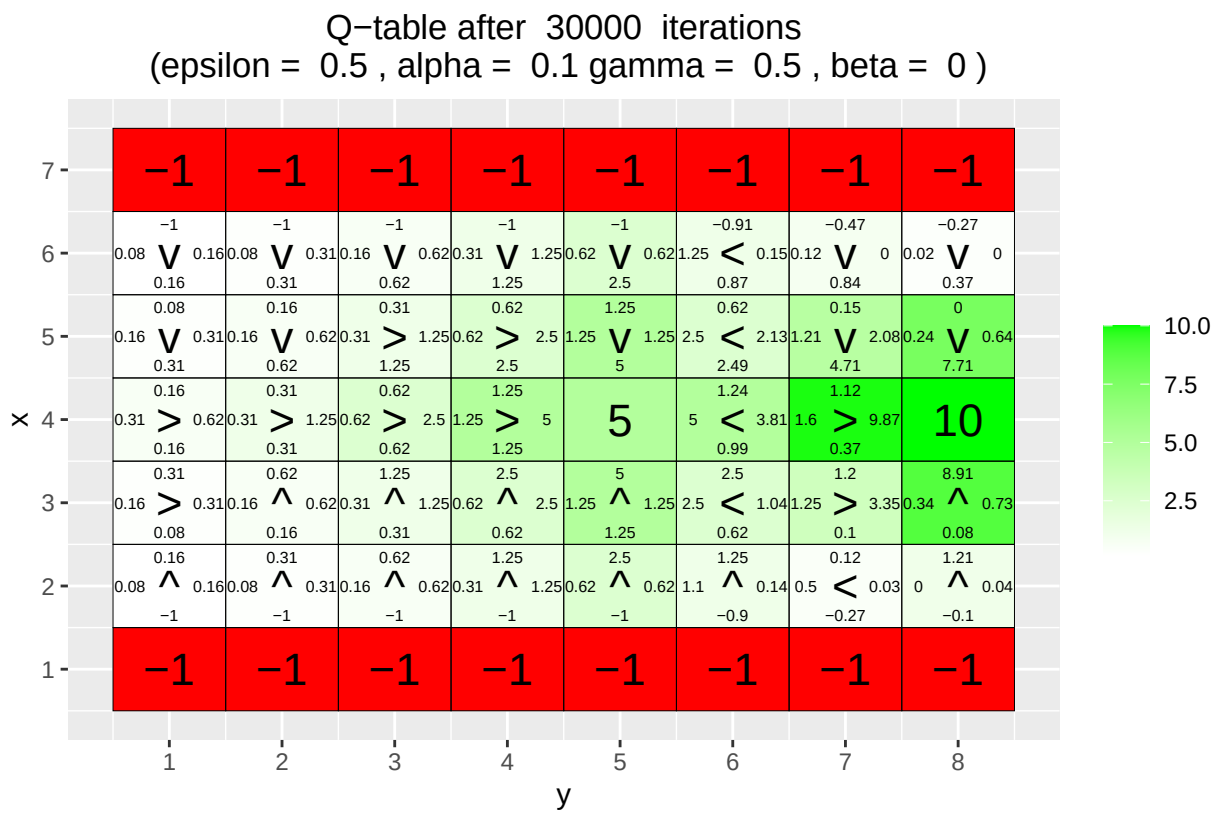
After 10000 episodes, the agent has learnt to entirely avoid the terminal states with negative rewards and always get to the terminal state with positive reward. But the final greedy policy is not in fact optimal. An optimal policy is one that maximizes the state value or the expected discounted return at every state s that the agent is in. This means that dropped in any state s, the agent should find a way to reach the desired terminal state in the most optimal way. This is true in this example for most of the states close to the terminal state, but there are few states away from the terminal states where the agent has not learnt optimality. This is because the q values for those states were not updated as often as for the other states (i.e. the agent explored that part of the search space less) and if the episodes are increased asymptotically, the final greedy policy would converge to the optimal policy.

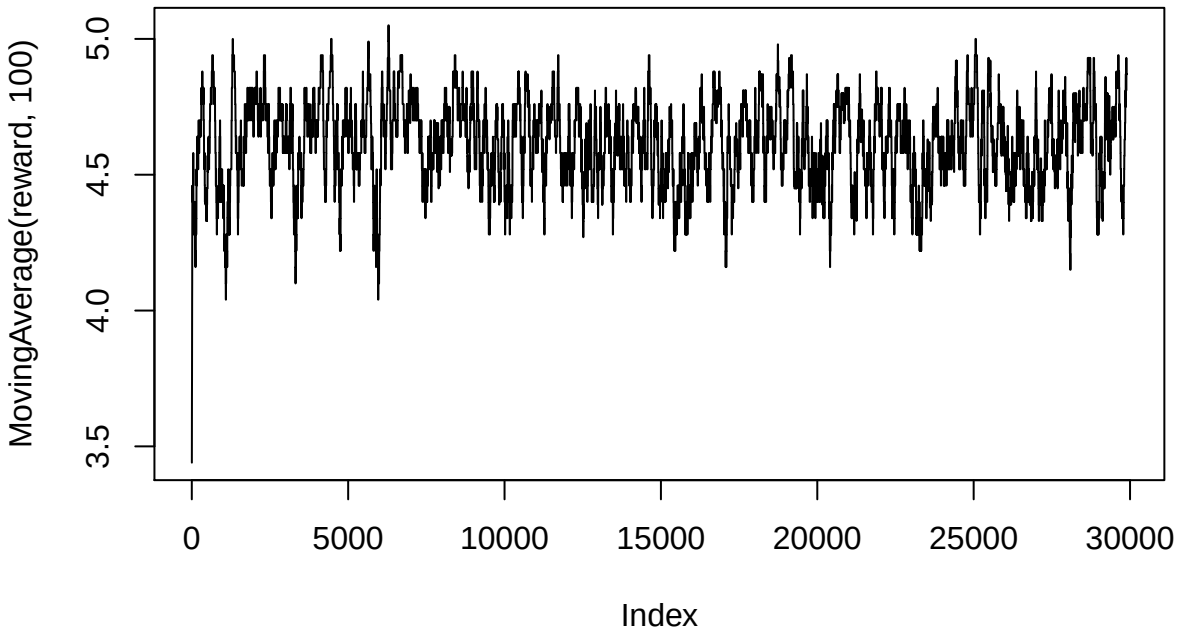
- Does the agent learn that there are multiple paths to get to the positive reward ? If not, what could be done to make the agent learn this ?

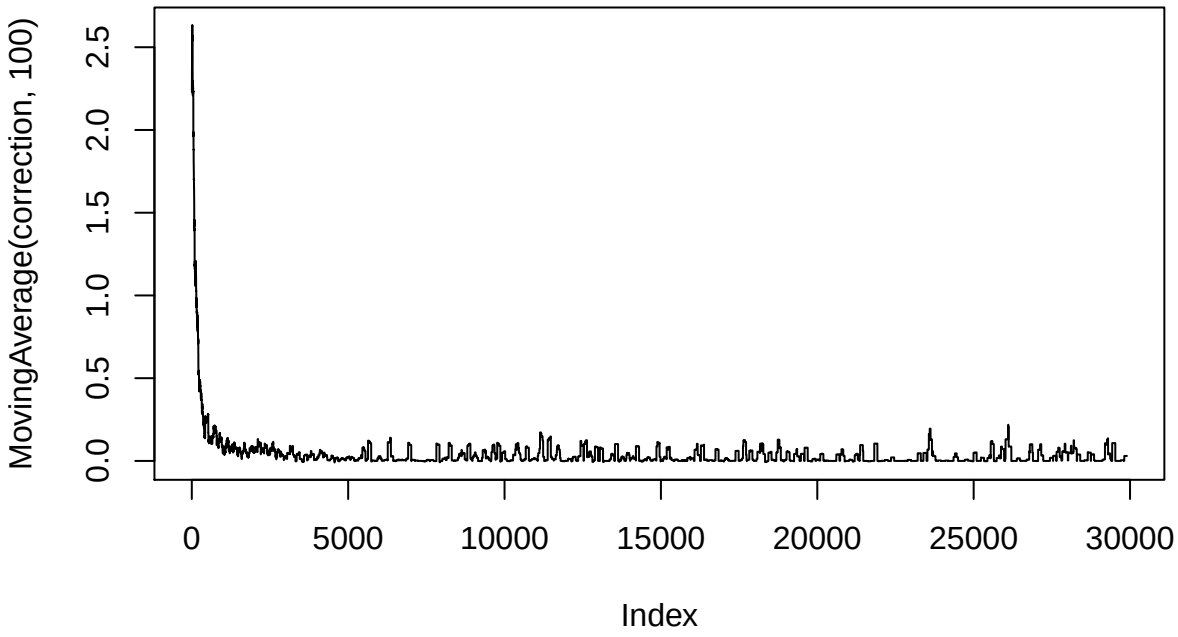
The agent has learnt multiple paths to get to the positive reward as indicated by the dense green heatmap around the positive reward state. All the states around it lead the agent to that terminal state. And the more the number of episodes, the more number of ways the agent will know how to reach the desired terminal state.

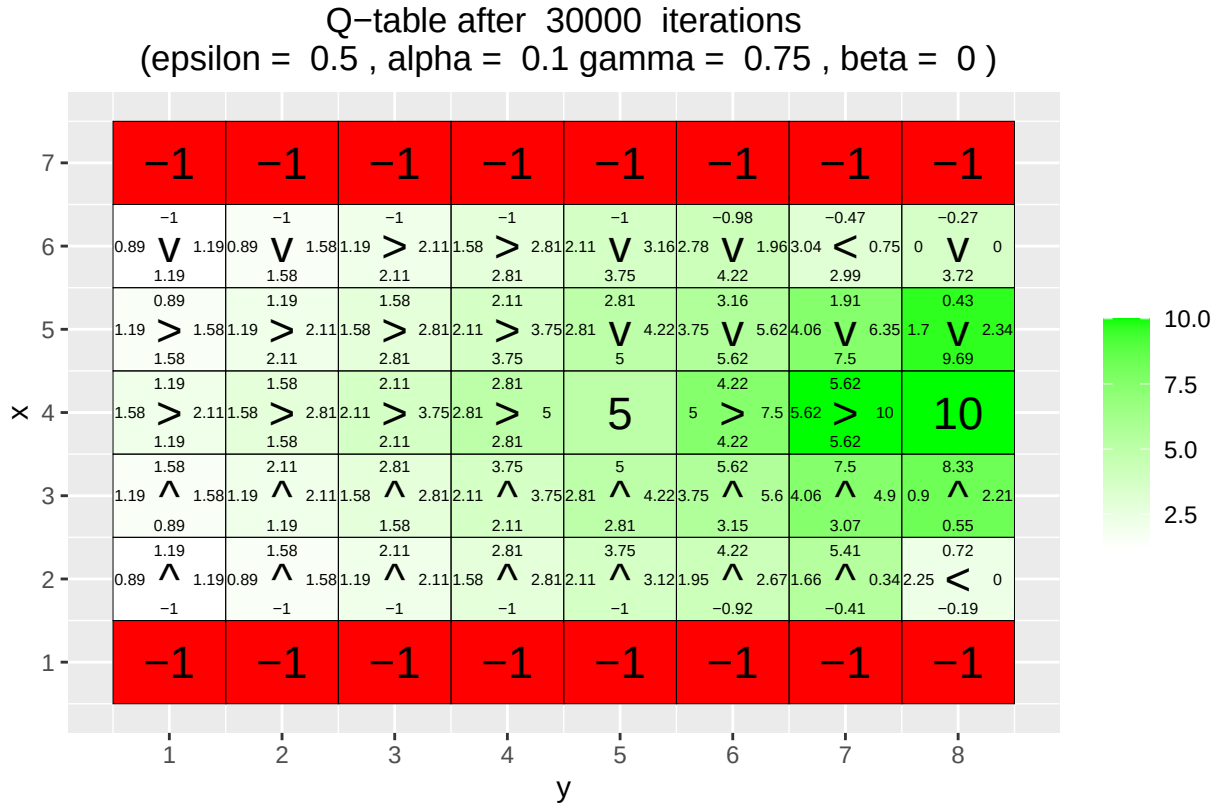
Environment B

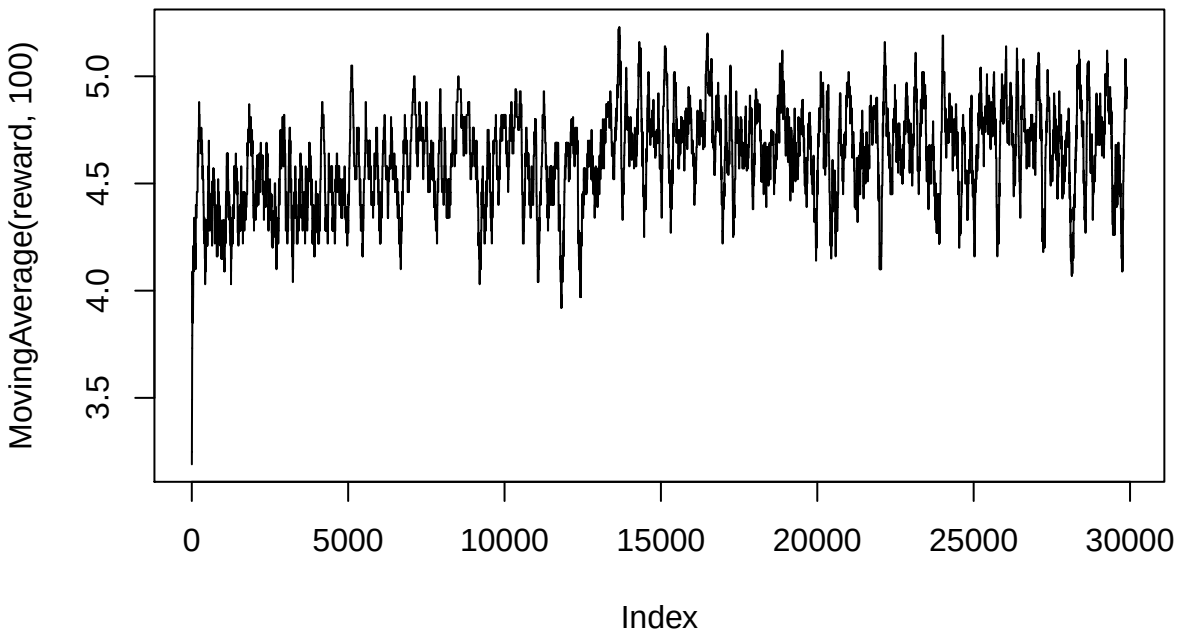


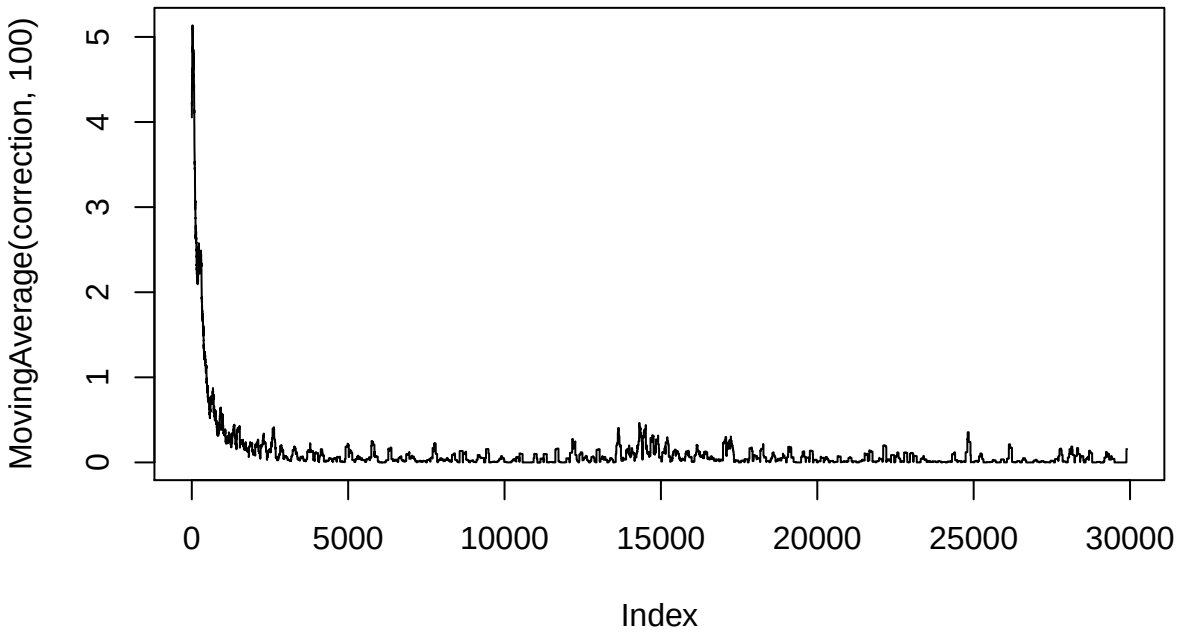


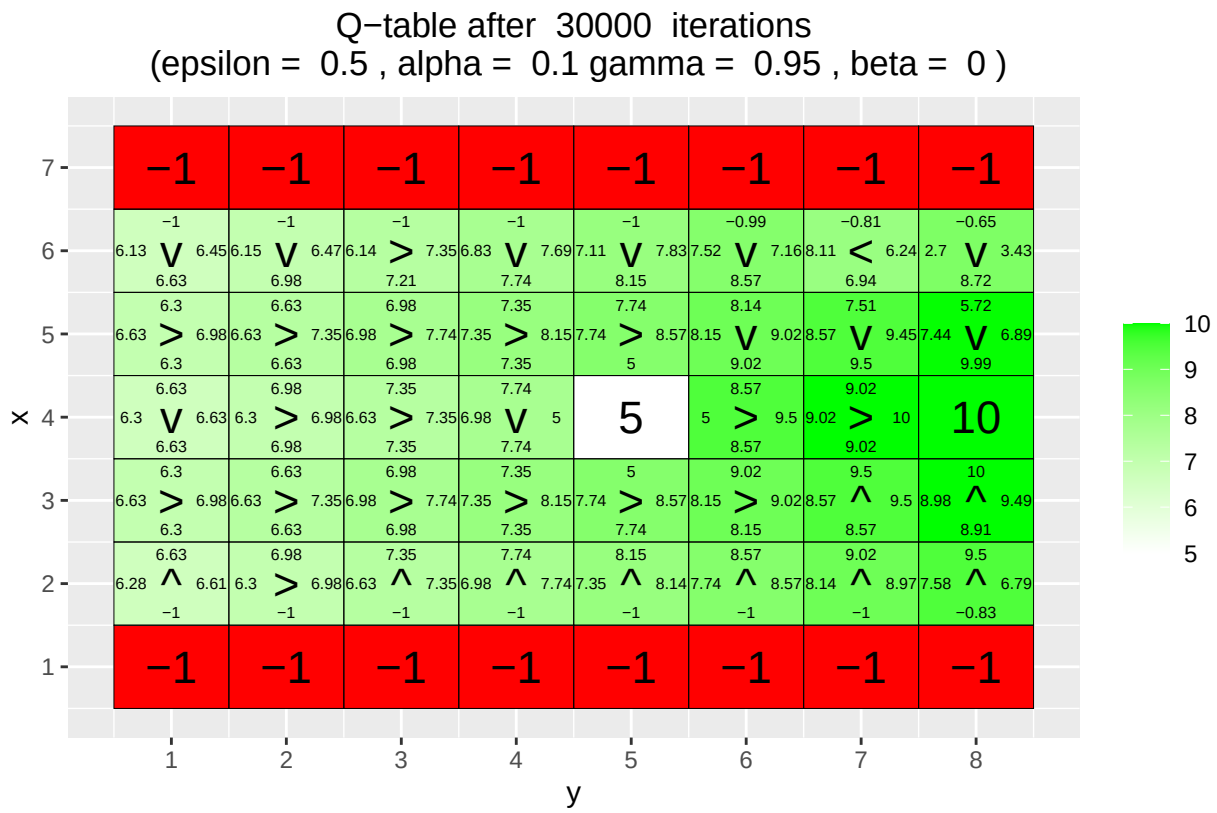

MovingAverage(reward, 100)
3.5
4.0
4.5
5.0
0
5000
10000
15000
20000
25000
30000
Index

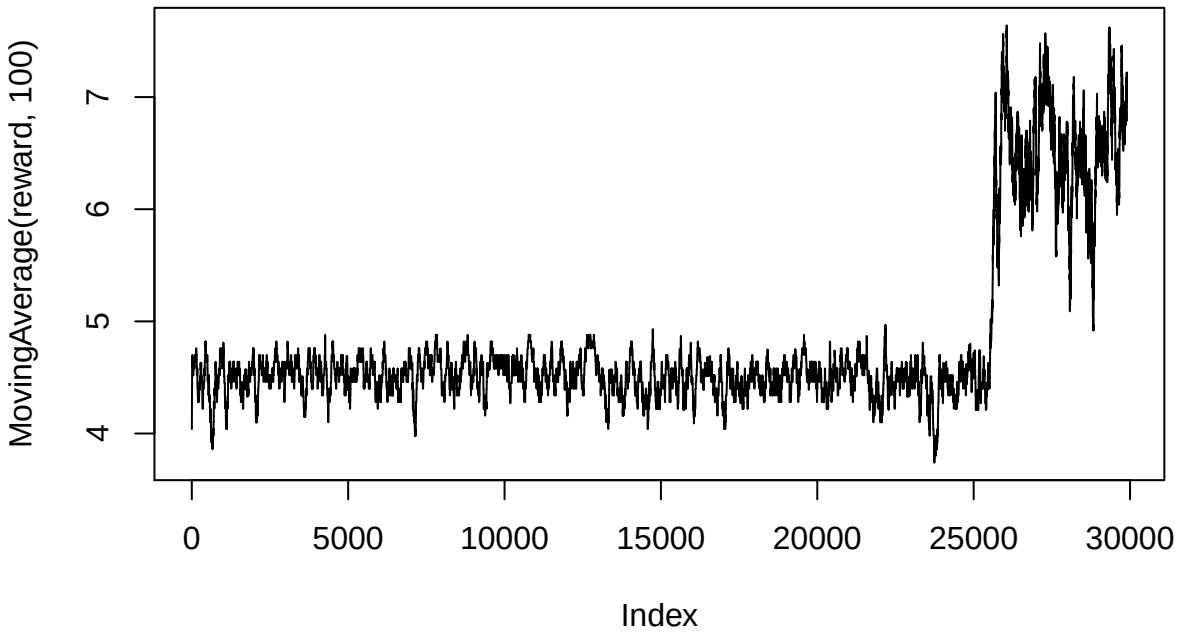

MovingAverage(correction, 100)
0.0
0.5
1.0
1.5
2.0
2.5
0
5000
10000
15000
20000
25000
30000
Index

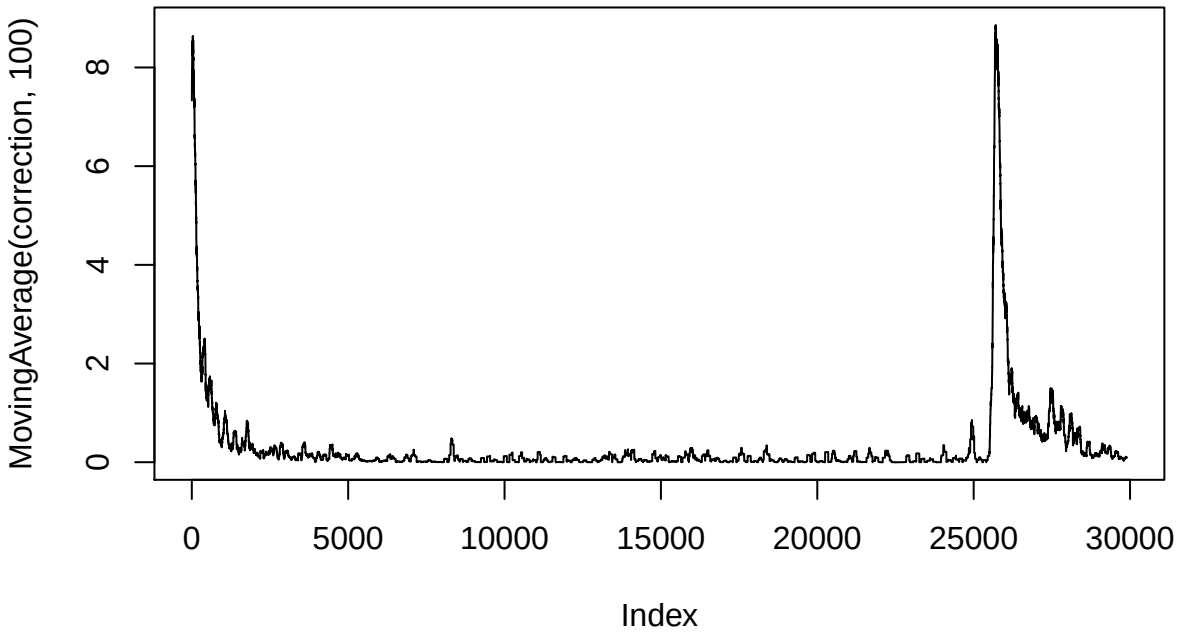

Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y
10.0
7.5
5.0
2.5


MovingAverage(reward, 100)
Index


MovingAverage(correction, 100)
0
1
2
3
4
5
0
5000
10000
15000
20000
25000
30000
Index


Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10
9
8
7
6
5


MovingAverage(reward, 100)
4
5
6
7
0
5000
10000
15000
20000
25000
30000
Index


MovingAverage(correction, 100)
0
2
4
6
8
0
5000
10000
15000
20000
25000
30000
Index

Effect of gamma

Epsilon=0.5 Epsilon=0.5 allows the agent to explore the environment reasonably well. In any state, the agent performs a completely random action With 50% probability to better explore the space around it. Under this setting, placing high, medium and low preference on immediate rewards (gamma=0.5, 0.75, 0.95) has quite different effects on the policies learnt by the agent.

Gamma=0.5 For gamma=0.5, there is a strong preference for rewards closer in time than later and the agent learns an optimal policy accordingly. Even though it has discovered the state with the higher reward, it still chooses to go to the lower reward state more often because it is closer and the discount factor quickly diminishes the delayed returns from the higher reward state which is further away. Only when the agent has wandered sufficiently close to the high reward state, it ends up there.

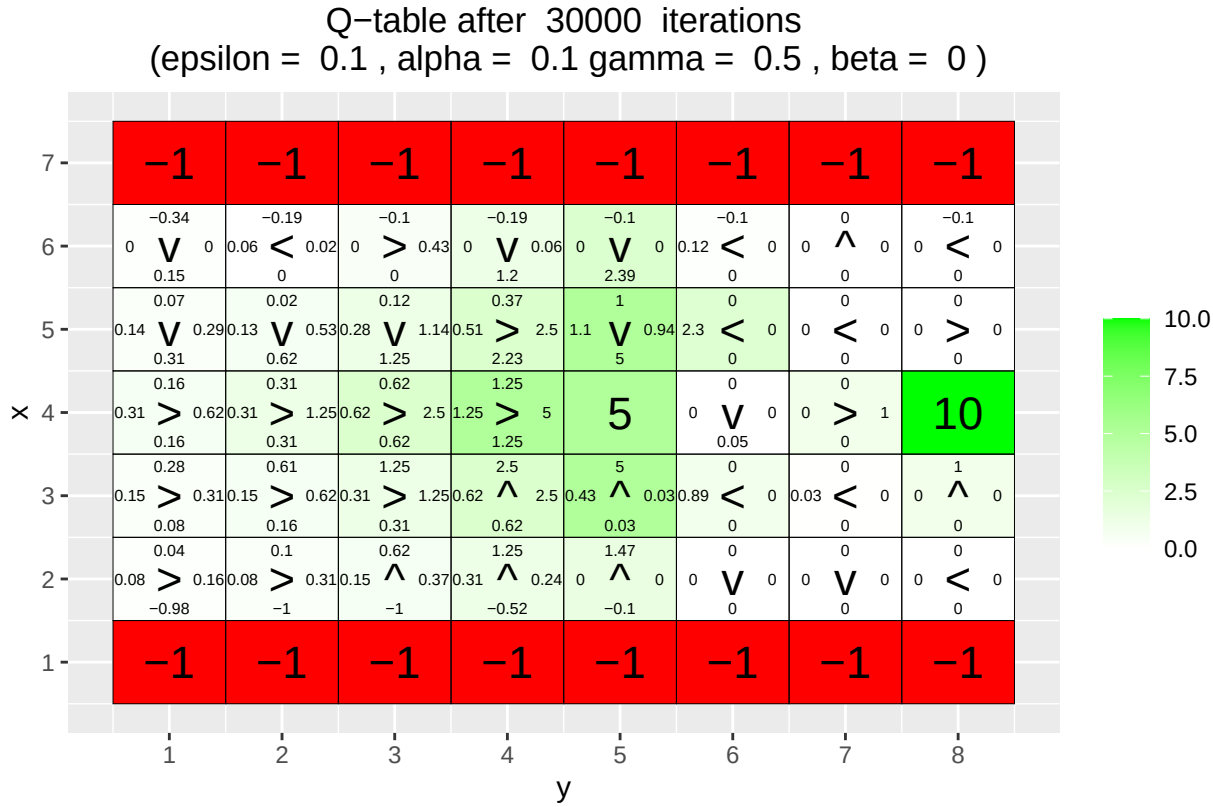
Gamma=0.75 For gamma=0.75, the agent learns a slightly different policy than the previous one. On reaching a state beyond the low reward state, it now chooses to move towards the high reward state more often rather than to quickly earn the lower reward. Now the expected discounted return for these states is high enough (the high reward is not totally offset by its distance) for the agent to end up in the higher reward state more frequently.

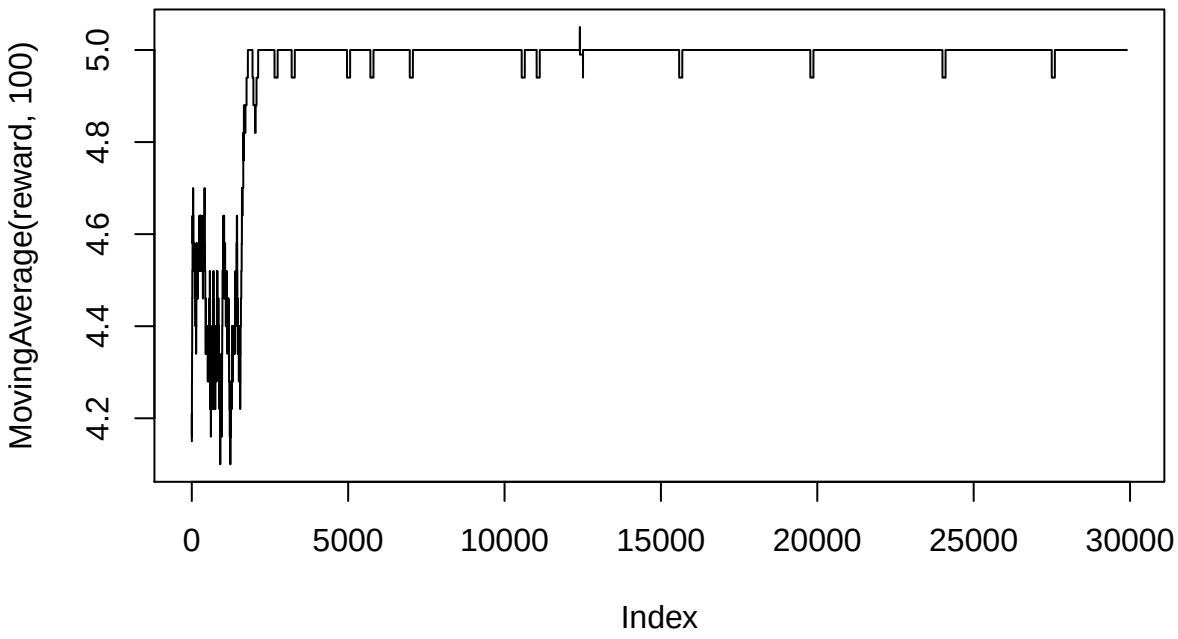
Gamma=0.95 For gamma=0.95, the agent learns to completely go around the low reward state to the high reward state. Now the delayed returns are as good as the immediate returns and the agent learns to entirely avoid the first terminal state even though it is closer because it discovers a much higher return by moving beyond it.

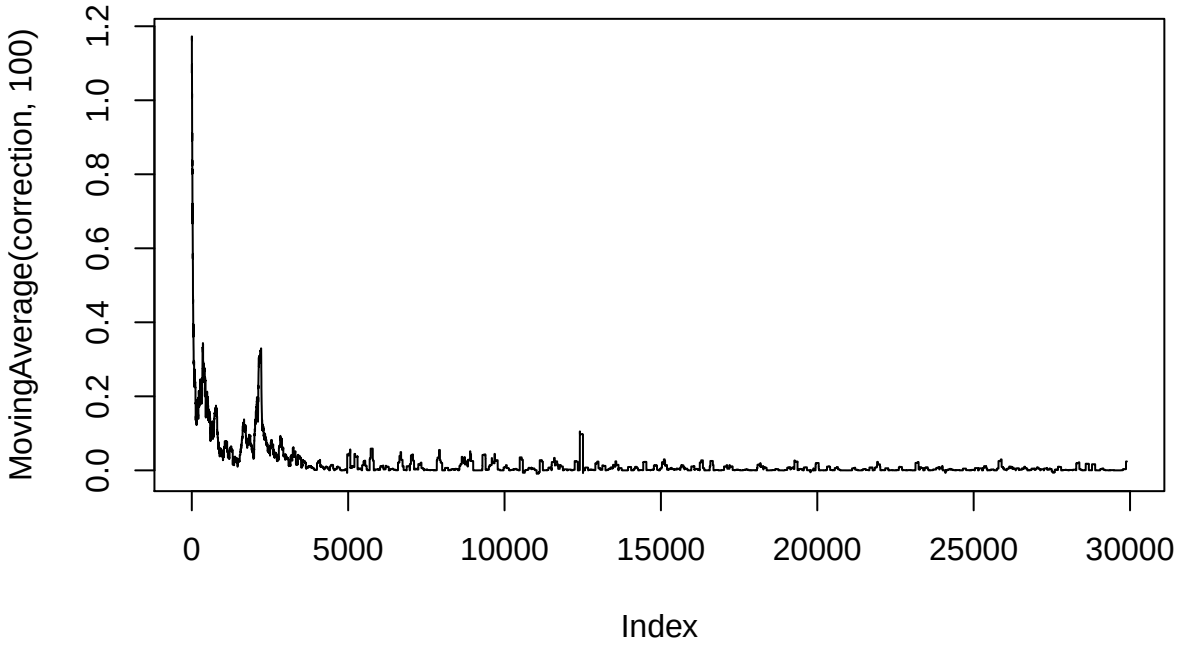
Moving Average plots

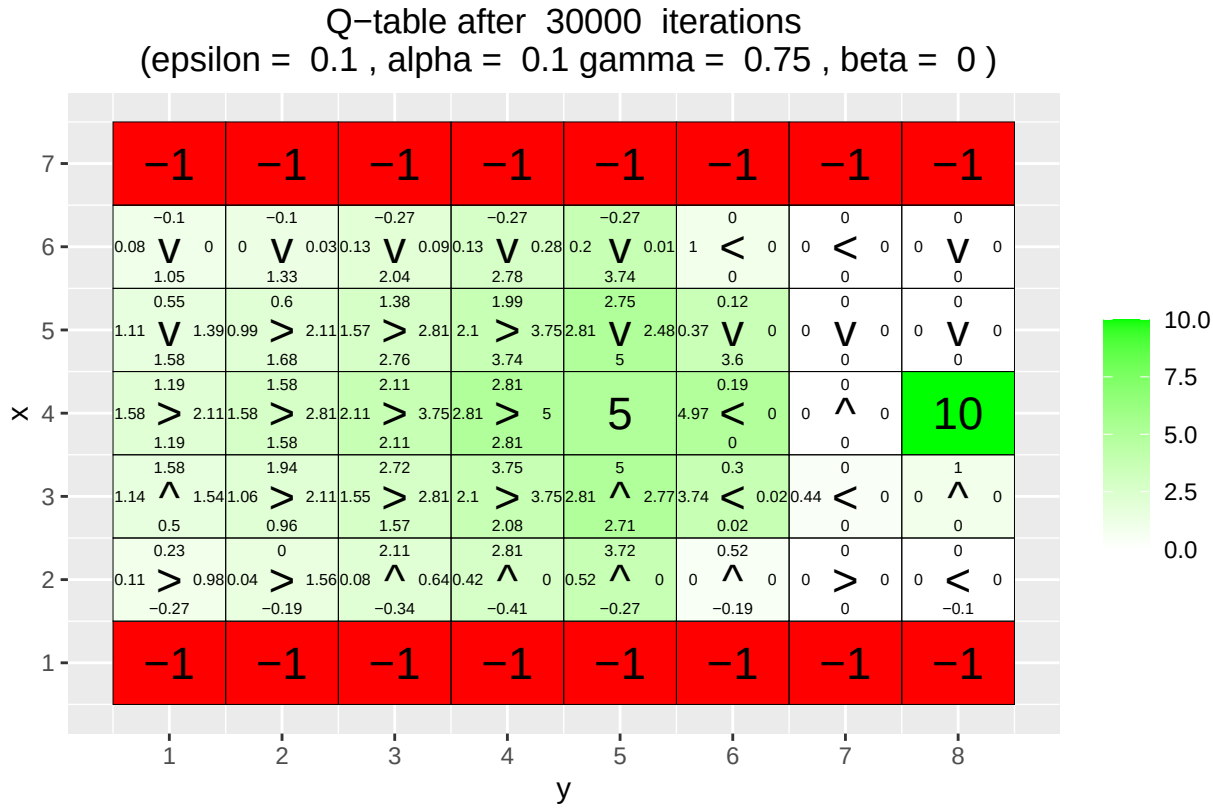
This is also highlighted in the moving average plots of the rewards and corrections. For gamma=0.5, 0.75, the rewards earned on an average is consistently between 4 and 5. This indicates that the agent did not end up in the high reward state too often under these settings or the average would have been higher. The average corrections are seen to be maximum in the first 1000 iterations or so. As the episodes increase, the correction average also drops close to 0. The agent becomes more confident of the Q-values learnt.

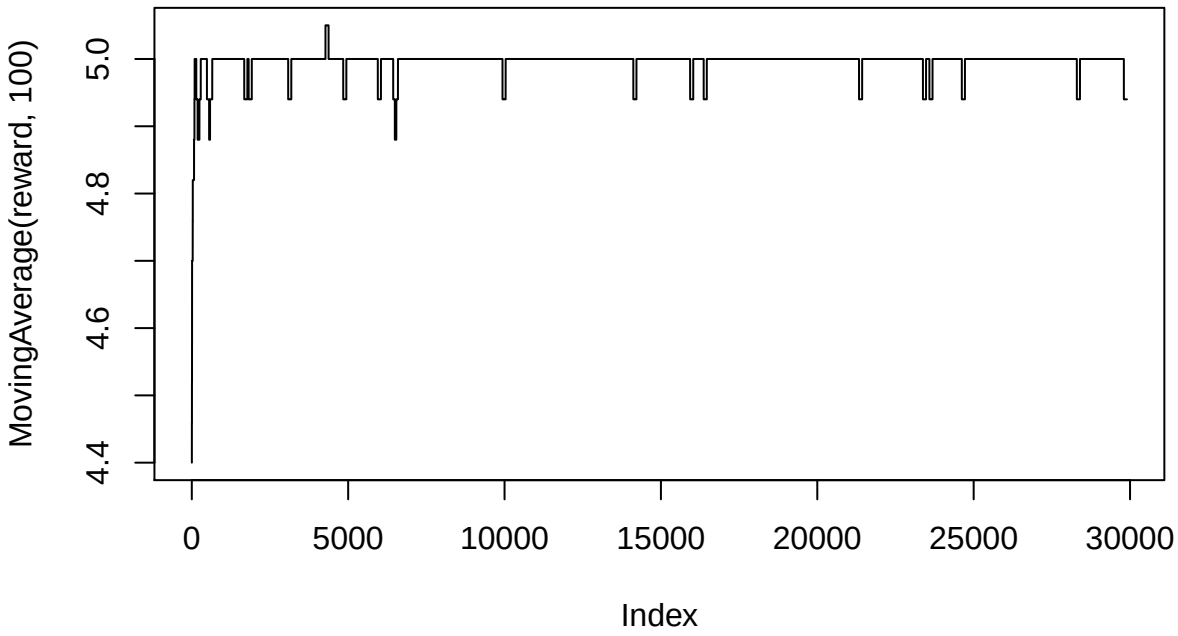
For gamma=0.95, the agent earns an average reward of 4.5 in the first 10-20 thousands of episodes and the average suddenly increases to 6.5 when it discovers the higher reward. The Q-values are then corrected and updated for all the states accordingly to maximize their expected return and we see a sharp increase in the correction average around the same time.

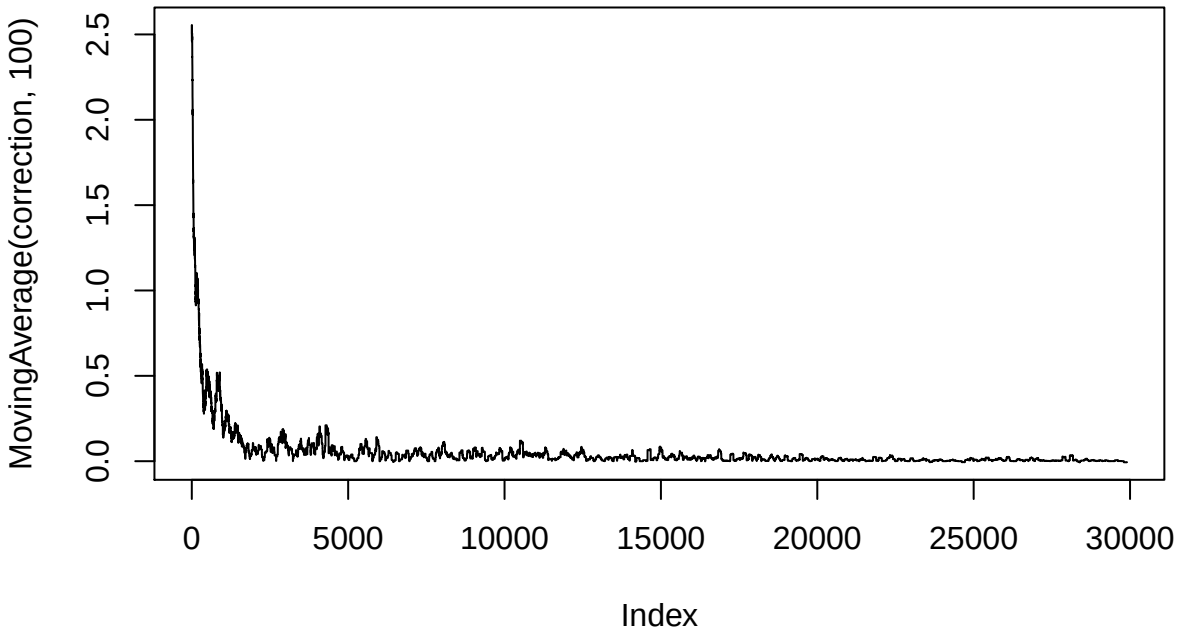

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.5 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0


MovingAverage(reward, 100)
Index

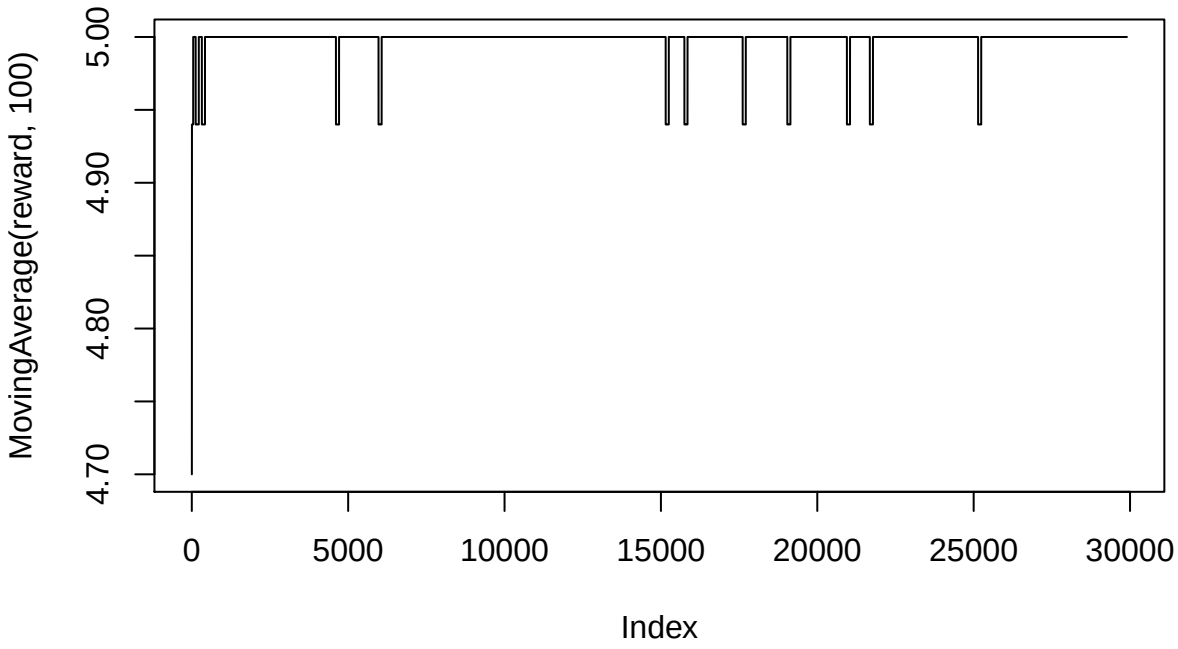

MovingAverage(correction, 100)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
5000
10000
15000
20000
25000
30000
Index

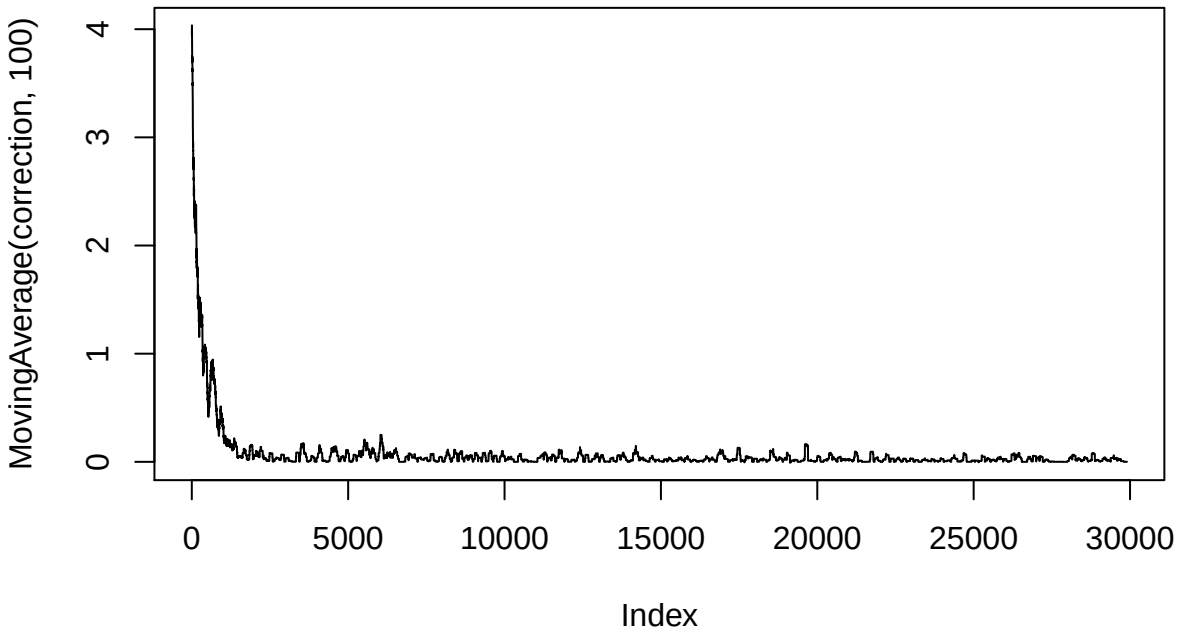

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0


MovingAverage(reward, 100)
4.4
4.6
4.8
5.0
0
5000
10000
15000
20000
25000
30000
Index


MovingAverage(correction, 100)
0.0
0.5
1.0
1.5
2.0
2.5
0
5000
10000
15000
20000
25000
30000
Index


Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0


MovingAverage(reward, 100)
Index



Effect of epsilon

Epsilon=0.1 Epsilon=0.1 is highly restrictive and forces the agent to follow the action with the maximal Q-value 90% of the time in any given state. Naturally, the environment is explored much more slowly under this setting and much of it is even left unexplored by the agent. The overall policies learnt for different values of gamma=0.5, 0.75, 0.95 are not very different in this setting. In all three cases, the agent learns to reach the reward state that is closer to it than go farther away looking for it.

Epsilon=0.5 is a better setting to evaluate the effect of gamma as it lets the agent to move freely allowing for the effects of gamma to be observed in. For epsilon=0.1, we need a lot more episodes to see the effect of gamma in the policies learnt by the agent.

Gamma=0.5 The low value of gamma hurries the agent to the closest reward while at the same time the low epsilon value restricts the agent from exploring too much in the current state.

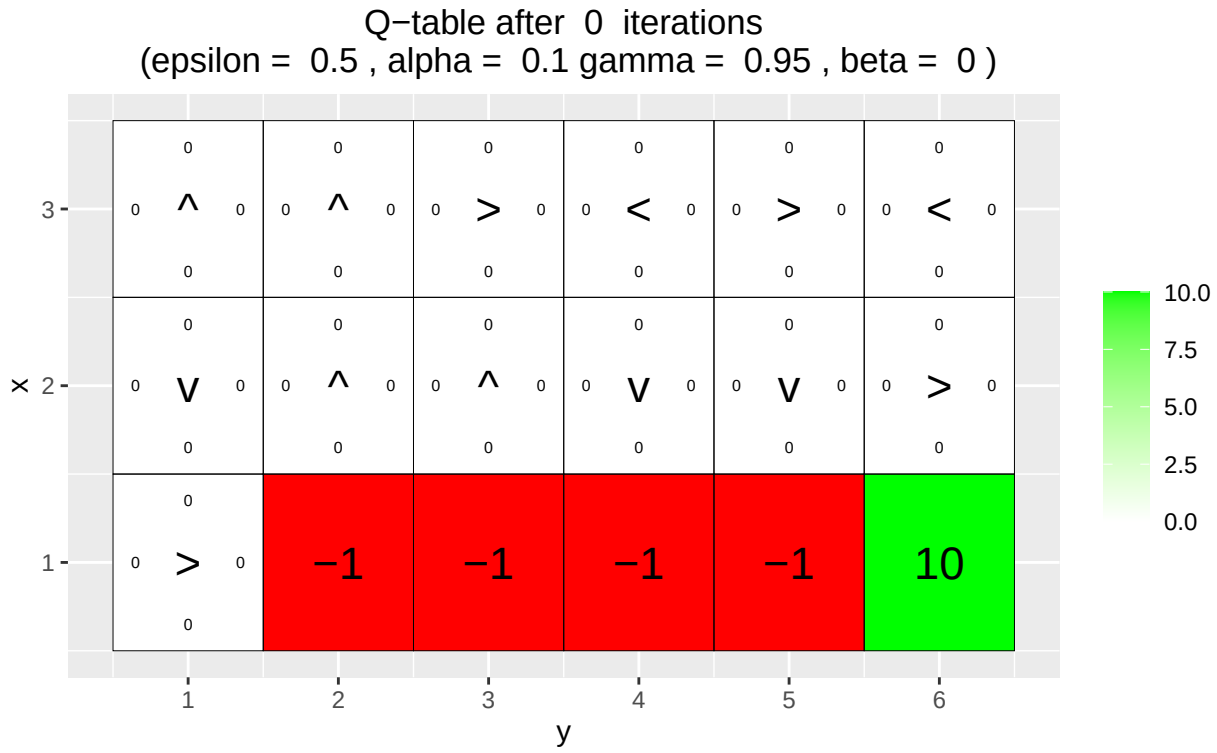
Gamma=0.75 Even though the agent is looking to maximize long term rewards, it is unable to go beyond the low reward state and find its way to the high reward state with the current Q-value updates and the non-randomness in its actions. It is exploring the environment much more restrictively and very slowly.

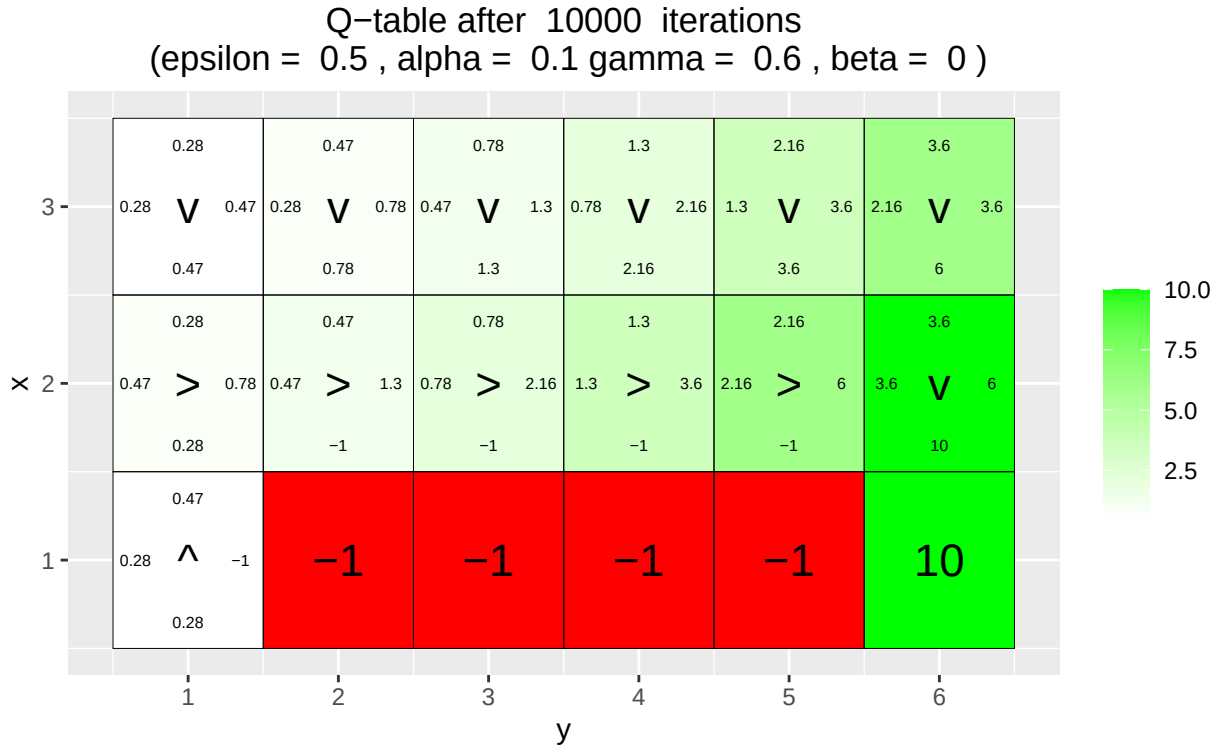
Gamma=0.95 Similarly here, the agent wants to maximize the long term rewards. But it requires a lot many more episodes to explore the environment beyond the low reward state and update the Q-table sufficiently to learn an optimal policy to reach the high reward state.

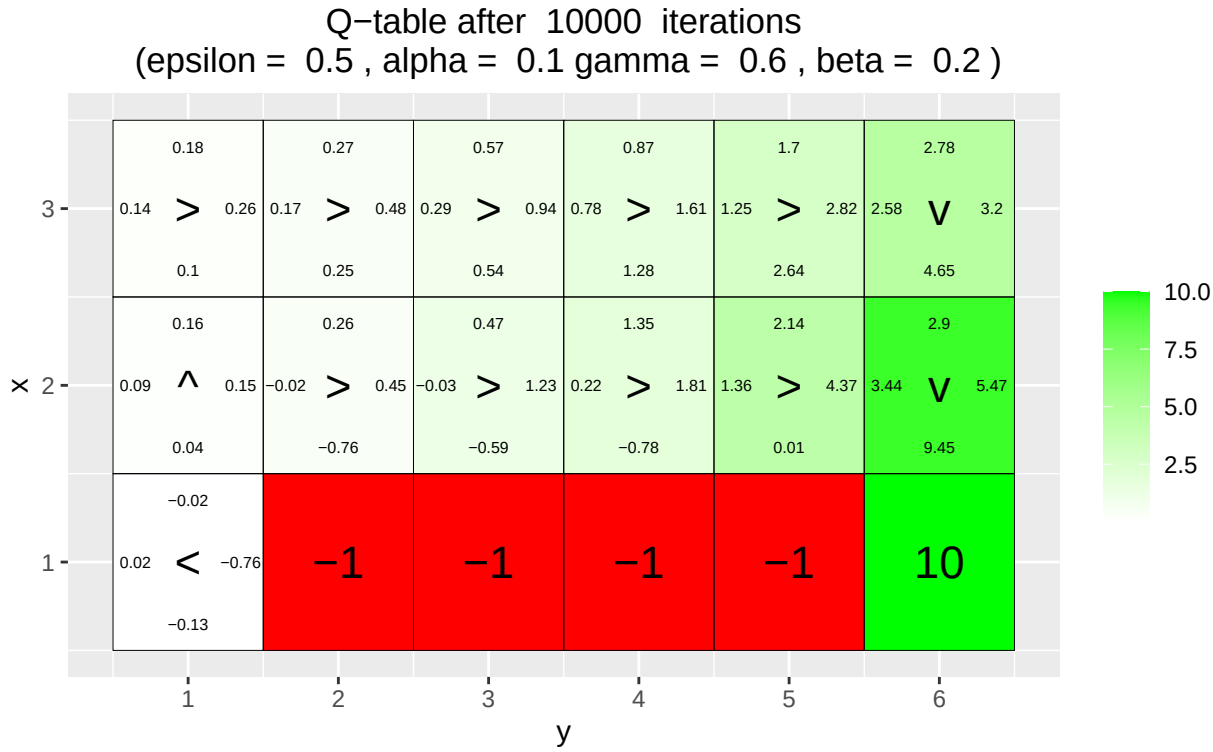
Moving Average plots

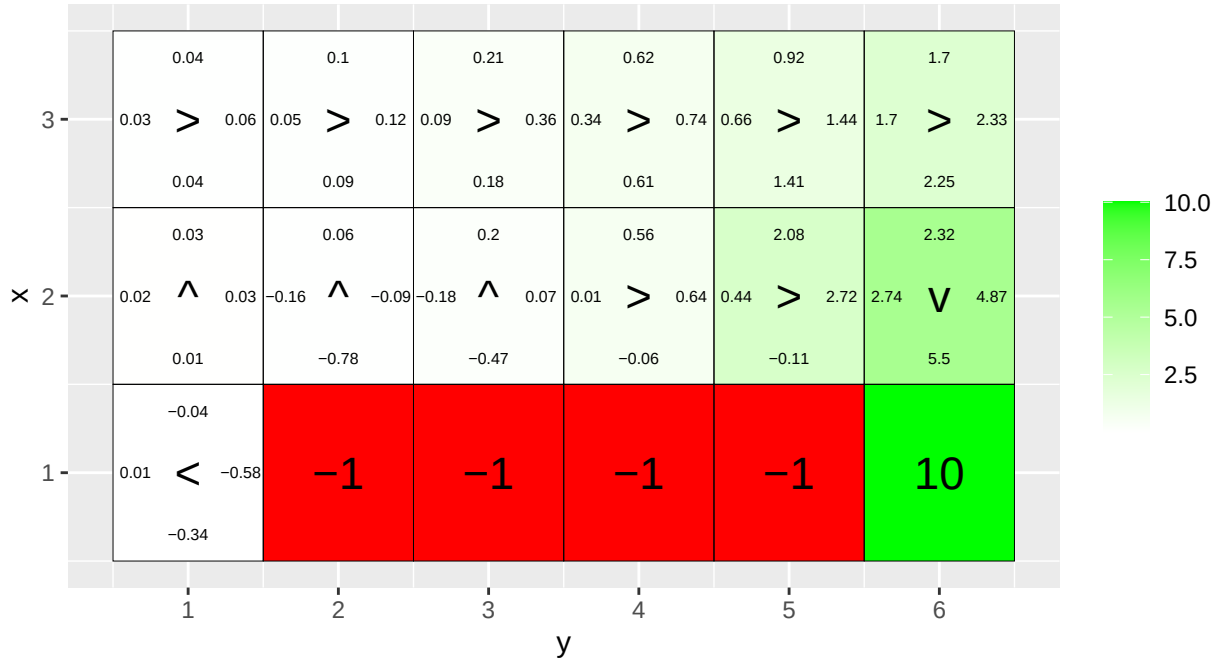
We see the moving average of rewards is much more consistently 5 in this setting of epsilon for all values of gamma. As the only terminal states it primarily explores are the low reward state and the negative terminal states.

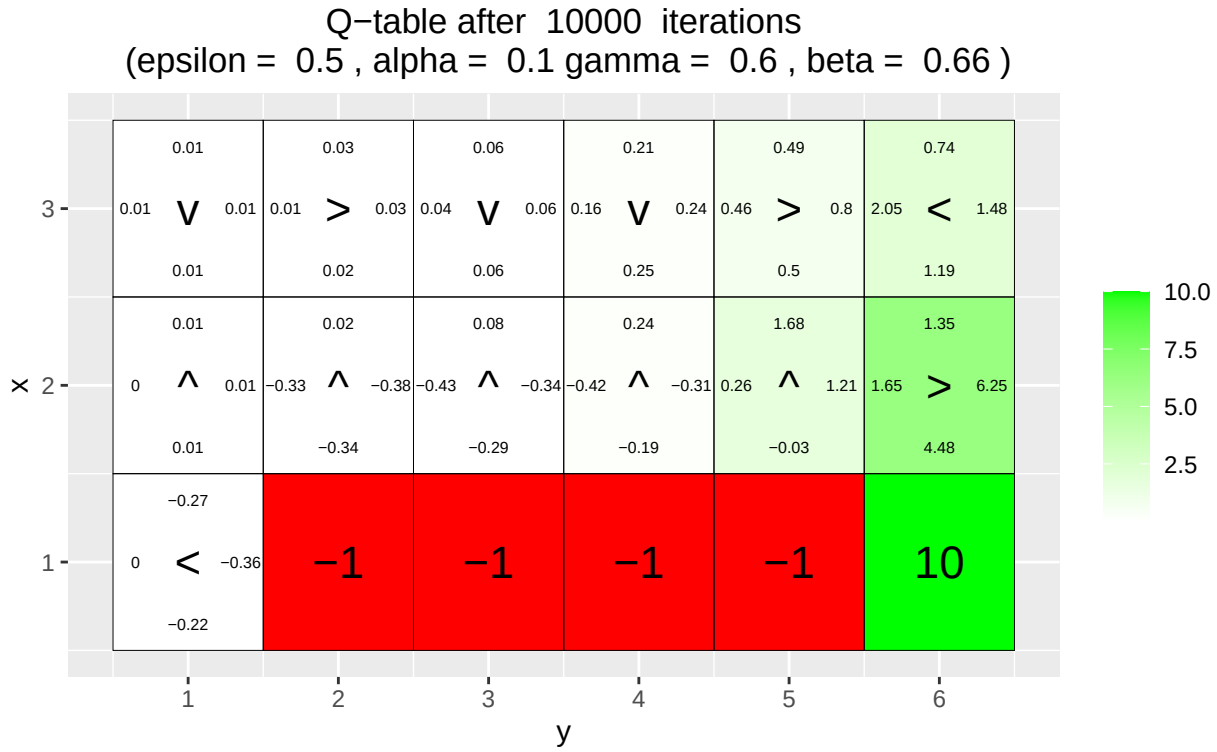
Environment C






Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.2)
0.18 0.14 > 0.26 0.1
0.27 0.17 > 0.48 0.25
0.57 0.29 > 0.94 0.54
0.87 0.78 > 1.61 1.28
1.7 1.25 > 2.82 2.64
2.78 2.58 v 3.2 4.65
0.16 0.09 ^ 0.15 0.04
0.26 −0.02 > 0.45 −0.76
0.47 −0.03 > 1.23 −0.59
1.35 0.22 > 1.81 −0.78
2.14 1.36 > 4.37 0.01
2.9 3.44 v 5.47 9.45
−0.02 0.02 < −0.76 −0.13
−1
−1
−1
−1
10
x
y
1
2
3
4
5
6
10.0
7.5
5.0
2.5


Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.4)
0.04 0.03 > 0.06 0.04
0.1 0.05 > 0.12 0.09
0.21 0.09 > 0.36 0.18
0.62 0.34 > 0.74 0.61
0.92 0.66 > 1.44 1.41
1.7 1.7 > 2.33 2.25
0.03 0.02 ^ 0.03 0.01
0.06 −0.16 ^ −0.09 −0.78
0.2 −0.18 ^ 0.07 −0.47
0.56 0.01 > 0.64 −0.06
2.08 0.44 > 2.72 −0.11
2.32 2.74 v 4.87 5.5
−0.04 0.01 < −0.58 −0.34
−1
−1
−1
−1
10
x
y
1
2
3
4
5
6
10.0
7.5
5.0
2.5

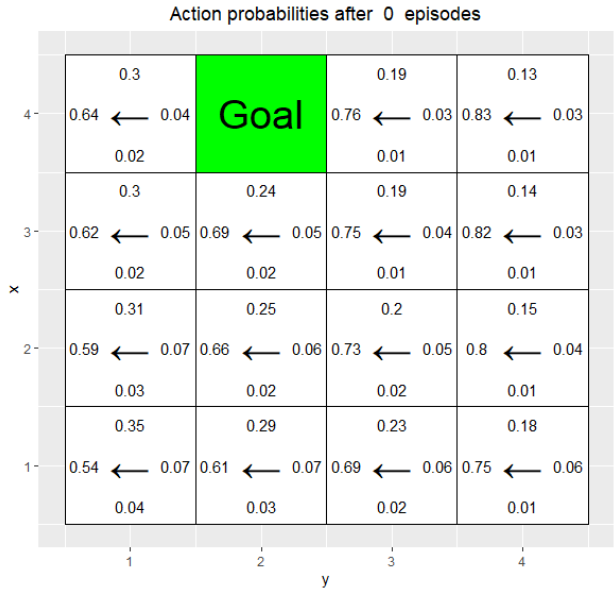


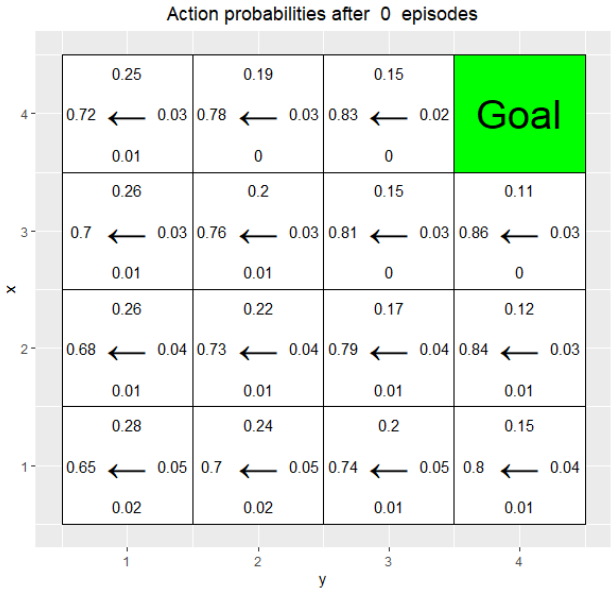
Effect of beta

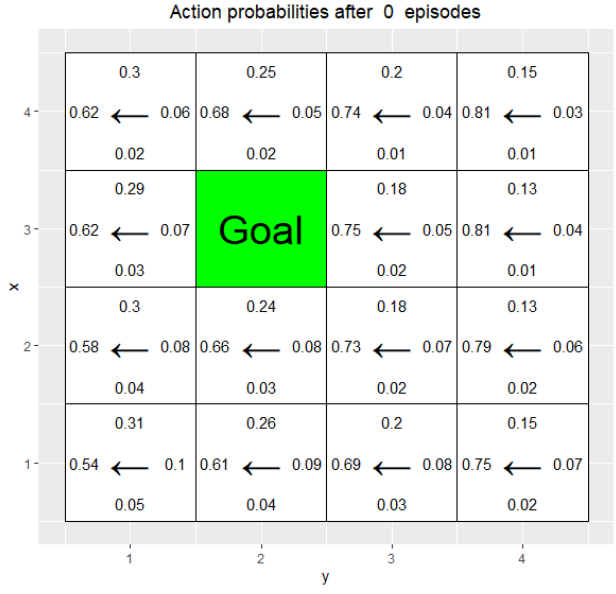
β is the probability of slipping to the left or right of the intended action. Naturally, $\beta = 0$ doesn't change the optimal policy learnt by the agent under a given setting of the other parameters i.e. the agent follows its intended actions with 100% probability. As the β value increases, we can see that the left and right action values for states close to the negative terminal states also get updated with negative value. The agent learns to be extra careful to avoid staying close to the negative terminal state (it tries to get a little bit extra further away from the negative terminal state) as the probability of slipping increases.

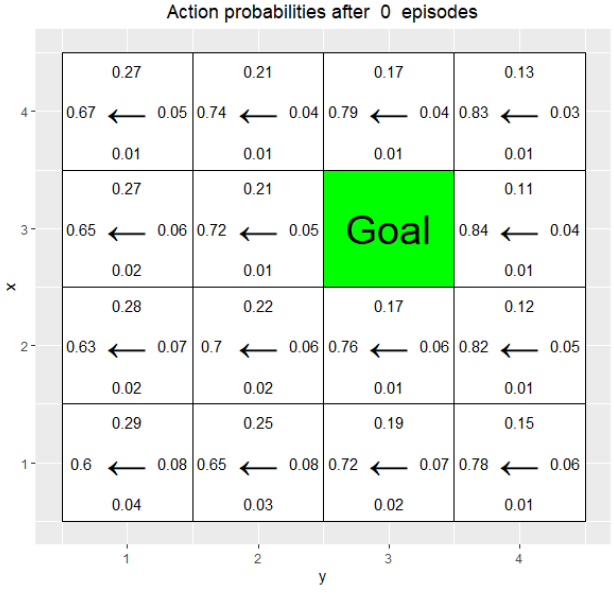
Reinforce

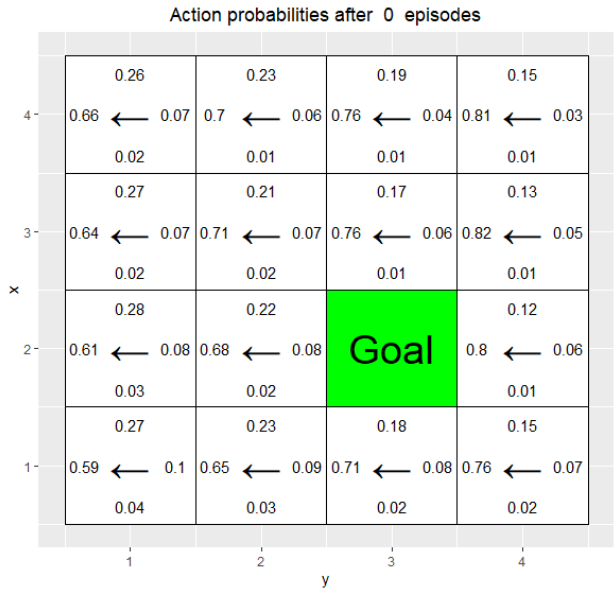
Environment D

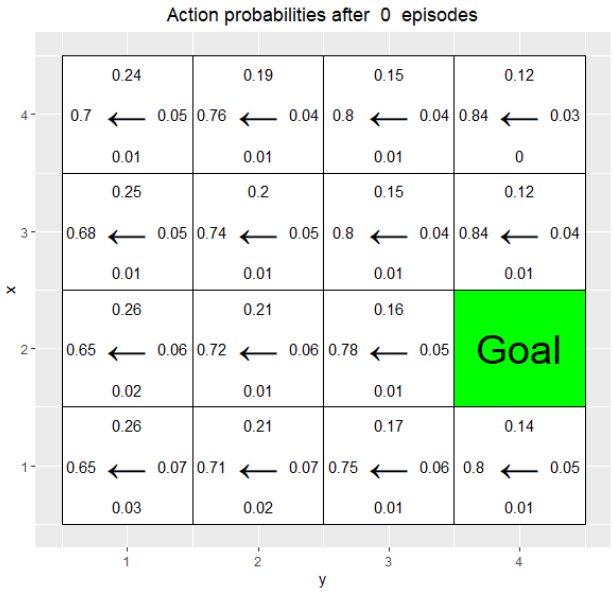


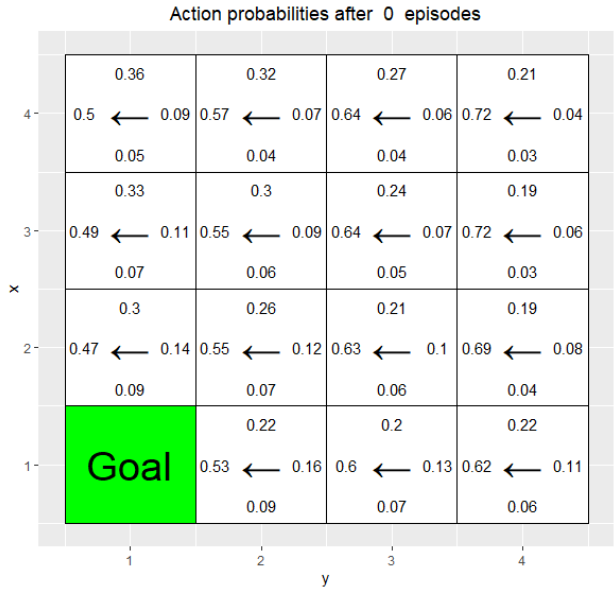


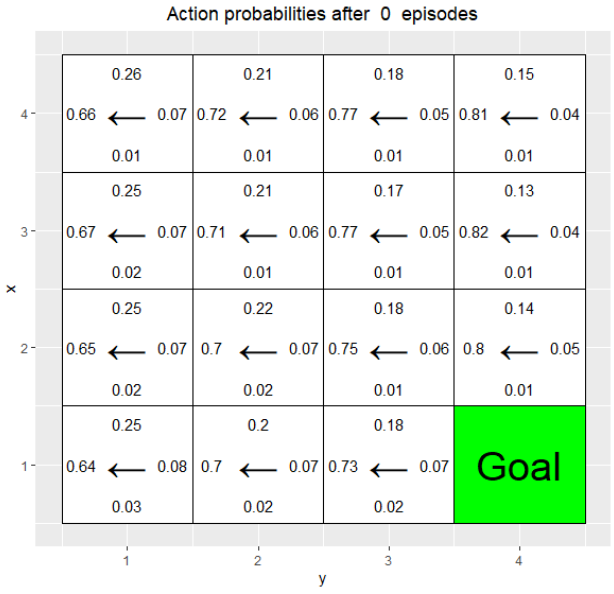


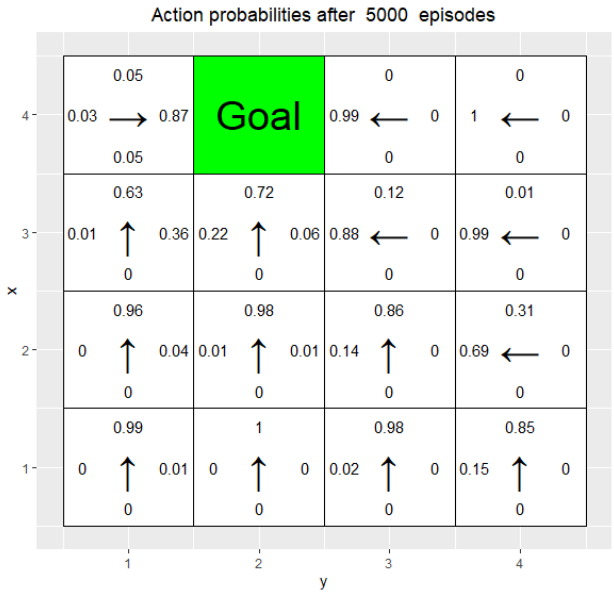


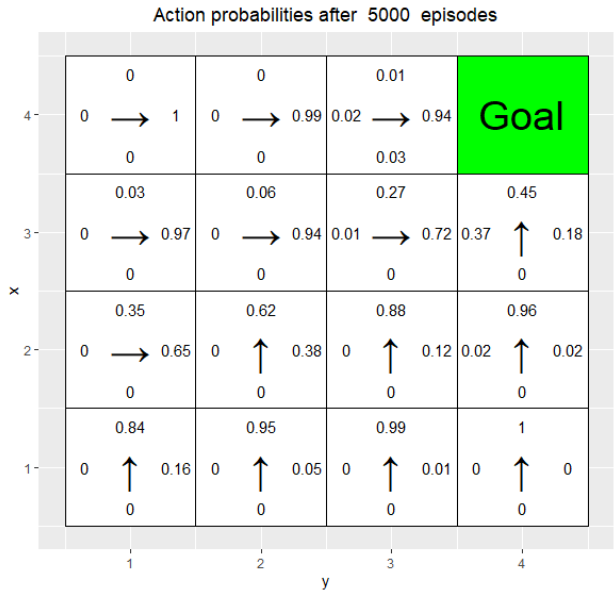

Action probabilities after 0 episodes
0.26
0.66 ← 0.07
0.02
0.23
0.7 ← 0.06
0.01
0.19
0.76 ← 0.04
0.01
0.15
0.81 ← 0.03
0.01
0.27
0.64 ← 0.07
0.02
0.21
0.71 ← 0.07
0.02
0.17
0.76 ← 0.06
0.01
0.13
0.82 ← 0.05
0.01
0.28
0.61 ← 0.08
0.03
0.22
0.68 ← 0.08
0.02
Goal
0.12
0.8 ← 0.06
0.01
0.27
0.59 ← 0.1
0.04
0.23
0.65 ← 0.09
0.03
0.18
0.71 ← 0.08
0.02
0.15
0.76 ← 0.07
0.02
x
y
1
2
3
4

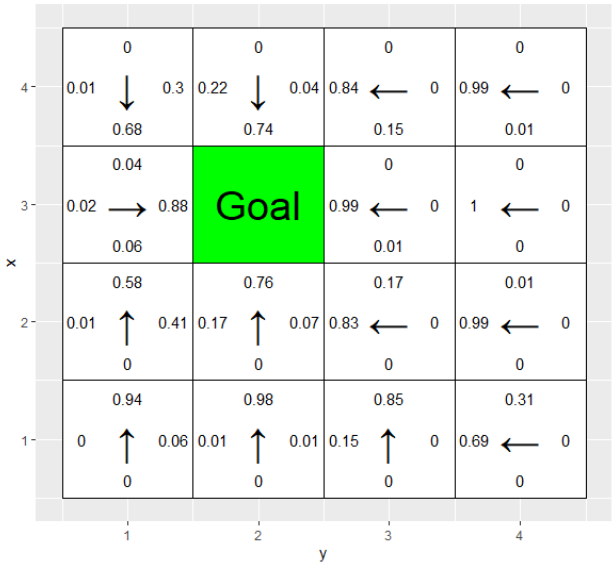

Action probabilities after 0 episodes
0.24
0.7 ← 0.05
0.01
0.19
0.76 ← 0.04
0.01
0.15
0.8 ← 0.04
0.01
0.12
0.84 ← 0.03
0
0.25
0.68 ← 0.05
0.01
0.2
0.74 ← 0.05
0.01
0.15
0.8 ← 0.04
0.01
0.12
0.84 ← 0.04
0.01
0.26
0.65 ← 0.06
0.02
0.21
0.72 ← 0.06
0.01
0.16
0.78 ← 0.05
0.01
Goal
0.26
0.65 ← 0.07
0.03
0.21
0.71 ← 0.07
0.02
0.17
0.75 ← 0.06
0.01
0.14
0.8 ← 0.05
0.01
x
y
1
2
3
4

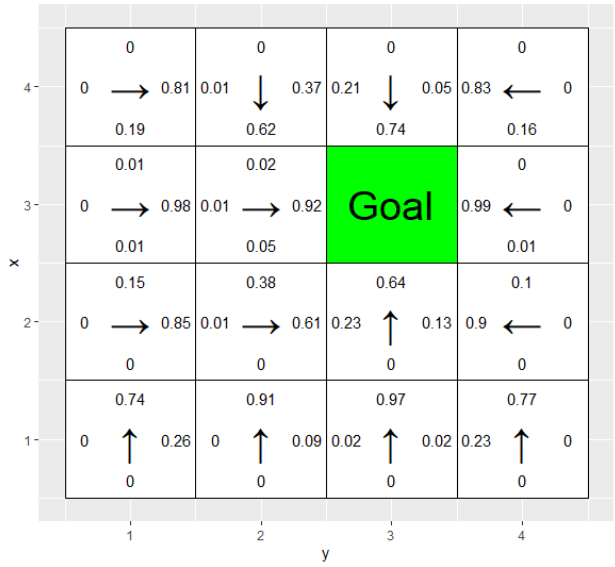

Action probabilities after 0 episodes
0.36
0.5 ← 0.09
0.05
0.32
0.57 ← 0.07
0.04
0.27
0.64 ← 0.06
0.04
0.21
0.72 ← 0.04
0.03
0.33
0.49 ← 0.11
0.07
0.3
0.55 ← 0.09
0.06
0.24
0.64 ← 0.07
0.05
0.19
0.72 ← 0.06
0.03
0.3
0.47 ← 0.14
0.09
0.26
0.55 ← 0.12
0.07
0.21
0.63 ← 0.1
0.06
0.19
0.69 ← 0.08
0.04
Goal
0.22
0.53 ← 0.16
0.09
0.2
0.6 ← 0.13
0.07
0.22
0.62 ← 0.11
0.06
x
y
1
2
3
4

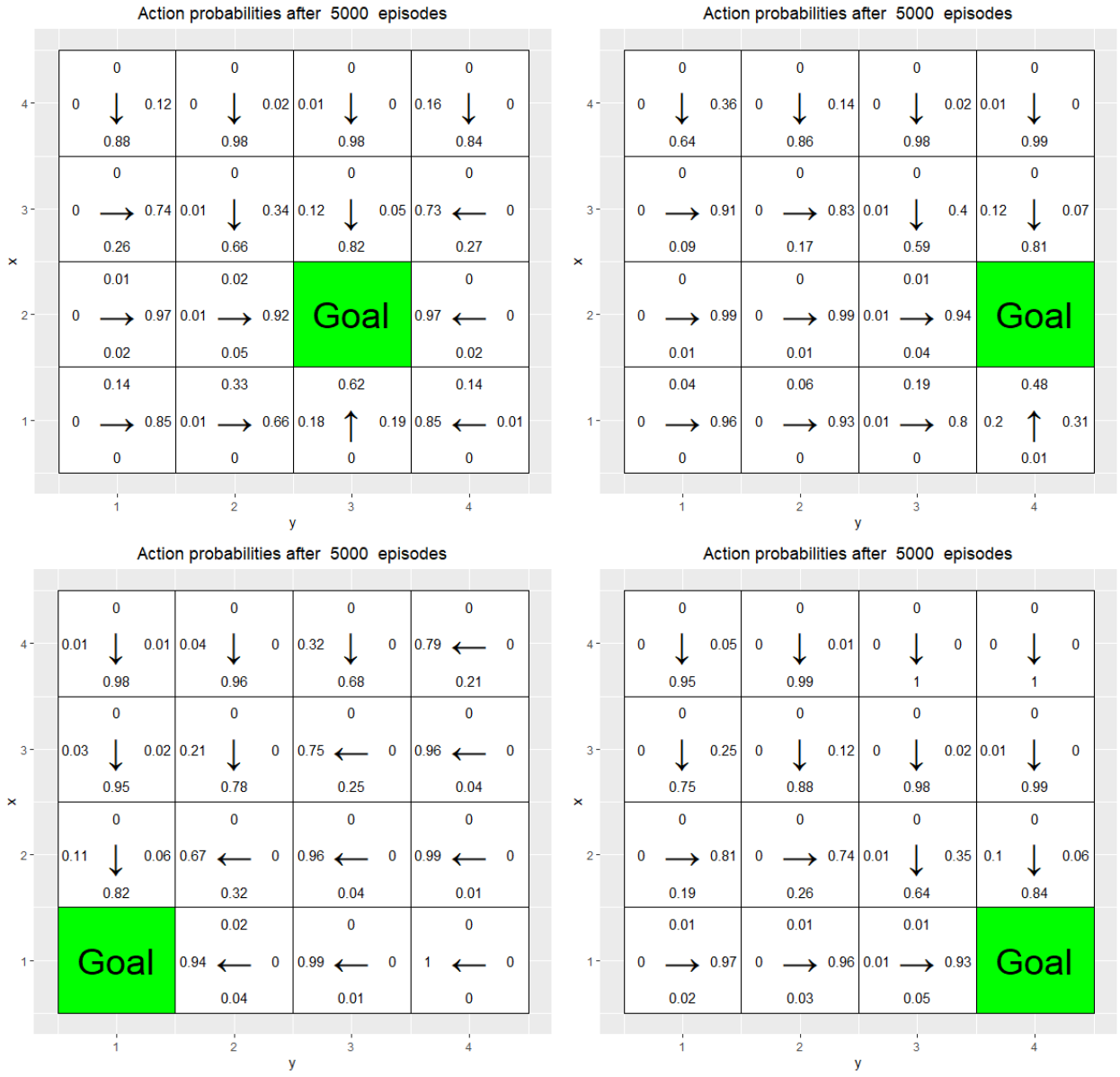

Action probabilities after 0 episodes
0.26
0.66 ← 0.07
0.01
0.21
0.72 ← 0.06
0.01
0.18
0.77 ← 0.05
0.01
0.15
0.81 ← 0.04
0.01
0.25
0.67 ← 0.07
0.02
0.21
0.71 ← 0.06
0.01
0.17
0.77 ← 0.05
0.01
0.13
0.82 ← 0.04
0.01
0.25
0.65 ← 0.07
0.02
0.22
0.7 ← 0.07
0.02
0.18
0.75 ← 0.06
0.01
0.14
0.8 ← 0.05
0.01
0.25
0.64 ← 0.08
0.03
0.2
0.7 ← 0.07
0.02
0.18
0.73 ← 0.07
0.02
Goal
x
y
1
2
3
4


Action probabilities after 5000 episodes
0.05
0.03 0.87
0.05
Goal
0 0.99 0
0
0 1 0
0
0.63
0.01 0.36
0
0.72
0.22 0.06
0
0.12
0.88 0 0
0.01
0.99 0 0
0.96
0 0.04
0
0.98
0.01 0.01
0
0.86
0.14 0 0
0.31
0.69 0 0
0.99
0 0.01
0
1
0 0 0
0.98
0.02 0 0
0.85
0.15 0 0
x
y


Action probabilities after 5000 episodes
0
0 1
0
0
0 0.99
0
0.01
0.02 0.94
0.03
Goal
0.03
0 0.97
0
0.06
0 0.94
0
0.27
0.01 0.72
0
0.45
0.37 0.18
0
0.35
0 0.65
0
0.62
0 0.38
0
0.88
0 0.12
0
0.96
0.02 0.02
0
0.84
0 0.16
0
0.95
0 0.05
0
0.99
0 0.01
0
1
0 0
0
x
y


Action probabilities after 5000 episodes
0
0.01 0.3
0.68
0
0.22 0.04
0.74
0
0.84 0 0.15
0
0.99 0 0.01
0.04
0.02 0.88
0.06
Goal
0
0.99 0 0.01
0
1 0 0
0.58
0.01 0.41
0
0.76
0.17 0.07
0
0.17
0.83 0 0
0.01
0.99 0 0
0.94
0 0.06
0
0.98
0.01 0.01
0
0.85
0.15 0 0
0.31
0.69 0 0
x
y


Action probabilities after 5000 episodes
0
0 0.81
0.19
0
0.01 0.37
0.62
0
0.21 0.05
0.74
0
0.83 0 0.16
0.01
0 0.98
0.01
0.02
0.01 0.92
0.05
Goal
0
0.99 0 0.01
0.15
0 0.85
0
0.38
0.01 0.61
0
0.64
0.23 0.13
0
0.1
0.9 0 0
0.74
0 0.26
0
0.91
0 0.09
0
0.97
0.02 0.02
0
0.77
0.23 0
0
x
y



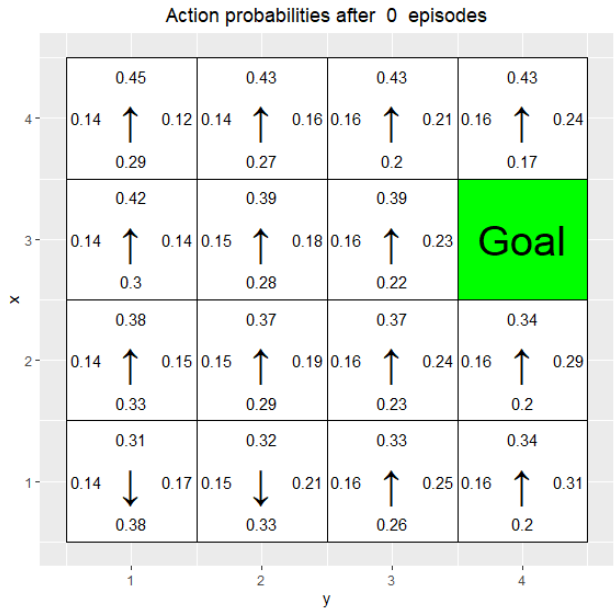
Has the agent learned a good policy? Why / Why not ?

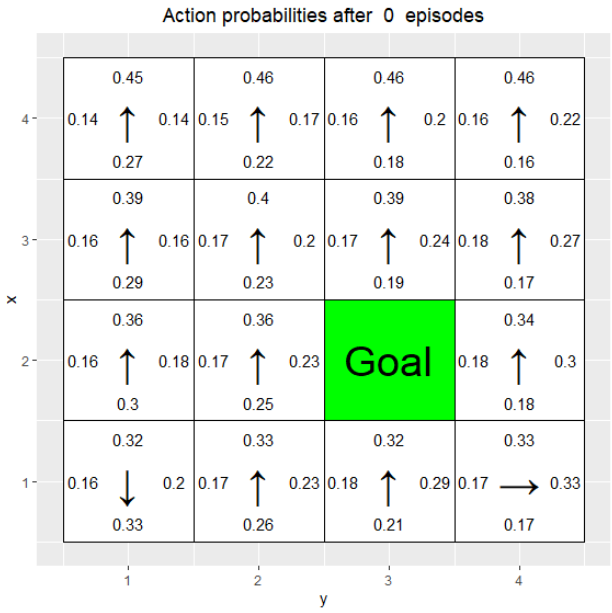
It seems like the agent has learnt a good policy since it generalizes well on goal states that it has never been trained on. Trough training, the agent has learnt a policy to reach the goal states used in training by optimizing the weight parameters and the Q-values conditioned on the parameters. The same policy when applied to completely new goal states during validation works very quite well as can be seen from the plots. For every validation goal state, the agent seems to have found an optimal path to reach it.

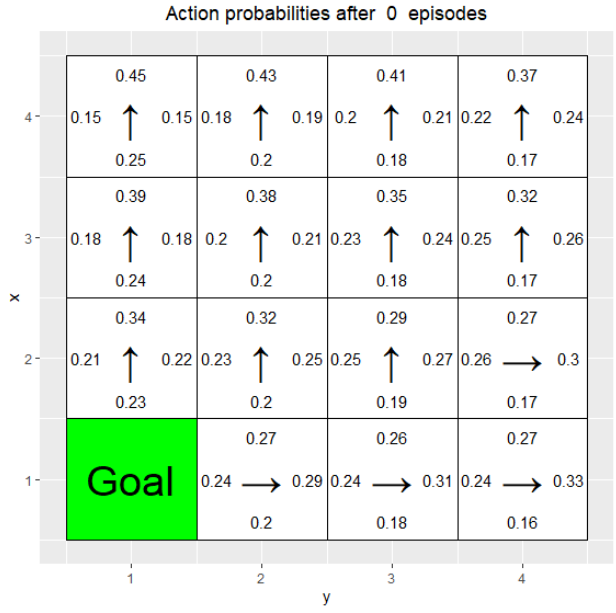
Could you have used the Q-learning algorithm to solve this task ?

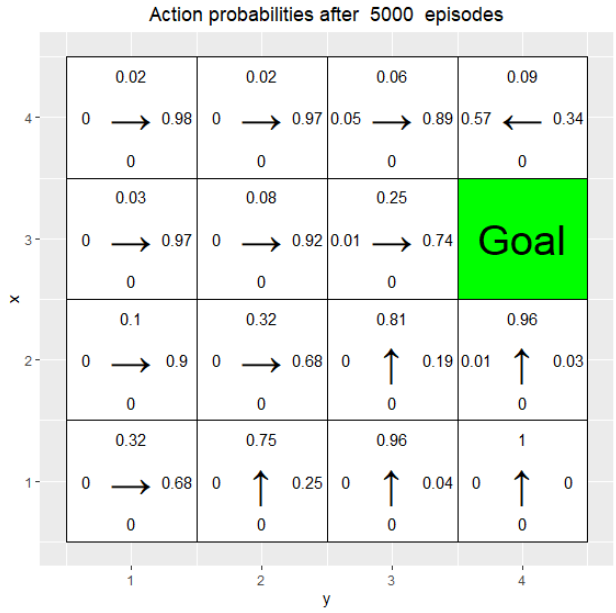
Q-learning could not have solved this task as it optimizes the actions (Q-values) for a specific reward map or environment. It is not the best suited for dynamically changing environments as the agent would need to be trained on every new environment. It does not generalize well like the REINFORCE algorithm.

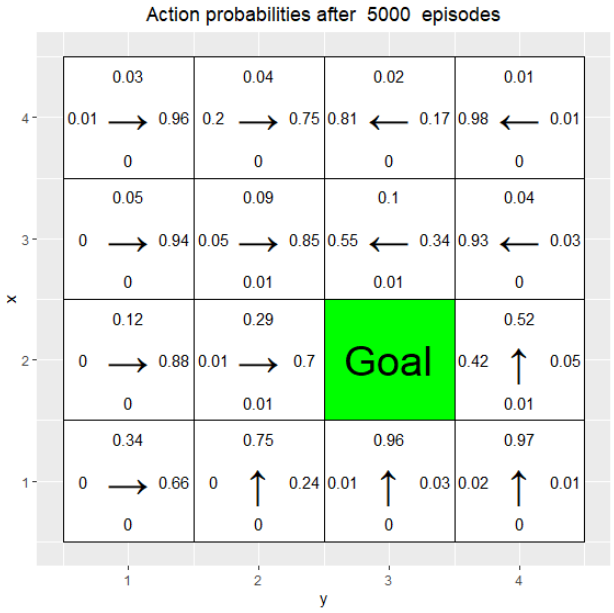
Environment E

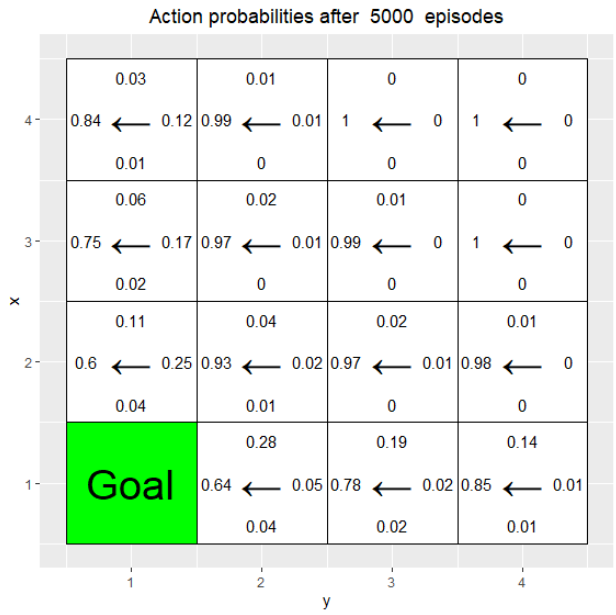












Has the agent learned a good policy? Why / Why not ?

The agent has not learnt a very good policy in this case. As seen from the plots, the agent doesn't really identify its new goal states or manage to always reach them. This is because the agent was trained only on top row goal positions and the optimal policy learnt was in a way biased to the sampling in the training data. The same policy didnt generalize well on other goal states during validation.

If the results obtained for environments D and E differ, explain why.

Even though it is the same algorithm, the taining samples were not generalized enough for the agent to learn a good policy (i.e. the algorithm did not converge to the optimal parameters that the policy is paramterized on). As in case of any NN, their results and predictions can only be interpolated and not extrapolated in the input value domain. So it can be said that in a way, the agent has only learnt to look for a goal state in the current row that it is in (except for when it randomly moves into a new row).

```r
knitr::opts_chunk$set(echo=TRUE, warning=FALSE, error=FALSE)
################################################################
# Q-learning
################################################################

# install.packages("ggplot2")
# install.packages("vctrs")
library(ggplot2)
library(vctrs)

# If you do not see four arrows in line 16, then do the following:
# File/Reopen with Encoding/UTF-8

arrows <- c("^", ">", "v", "<")
#arrows <- c("↑", "→", "↓", "←")
action_deltas <- list(c(1,0),  # up
                      c(0,1),  # right
                      c(-1,0), # down
                      c(0,-1)) # left

vis_environment <- function(iterations=0, epsilon = 0.5, alpha = 0.1, gamma = 0.95, beta = 0){

  # Visualize an environment with rewards.
  # Q-values for all actions are displayed on the edges of each tile.
  # The (greedy) policy for each state is also displayed.
  # ,0
  # Args:
  #   iterations, epsilon, alpha, gamma, beta (optional): for the figure title.
  #   reward_map (global variable): a HxW array containing the reward given at each state.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #   H, W (global variables): environment dimensions.

  df <- expand.grid(x=1:H,y=1:W)
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,1],NA),df$x,df$y)
  df$val1 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,2],NA),df$x,df$y)
  df$val2 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,3],NA),df$x,df$y)
  df$val3 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,4],NA),df$x,df$y)
  df$val4 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y)
    ifelse(reward_map[x,y] == 0,arrows[GreedyPolicy(x,y)],reward_map[x,y]),df$x,df$y)
  df$val5 <- as.vector(foo)
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,max(q_table[x,y,]),
                                     ifelse(reward_map[x,y]<0,NA,reward_map[x,y])),df$x,df$y)
  df$val6 <- as.vector(foo)

  print(ggplot(df,aes(x = y,y = x)) +
          scale_fill_gradient(low = "white", high = "green", na.value = "red", name = "") +
          geom_tile(aes(fill=val6)) +
          geom_text(aes(label = val1),size = 2,nudge_y = .35,na.rm = TRUE) +
          geom_text(aes(label = val2),size = 2,nudge_x = .35,na.rm = TRUE) +
```

```
          geom_text(aes(label = val3),size = 2,nudge_y = -.35,na.rm = TRUE) +
          geom_text(aes(label = val4),size = 2,nudge_x = -.35,na.rm = TRUE) +
          geom_text(aes(label = val5),size = 6) +
          geom_tile(fill = 'transparent', colour = 'black') +
          ggtitle(paste("Q-table after ",iterations," iterations\n",
                        "(epsilon = ",epsilon,", alpha = ",alpha,"gamma = ",gamma,", beta = ",beta,")")) +
          theme(plot.title = element_text(hjust = 0.5)) +
          scale_x_continuous(breaks = c(1:W),labels = c(1:W)) +
          scale_y_continuous(breaks = c(1:H),labels = c(1:H)))

}

GreedyPolicy <- function(x, y){

  # Get a greedy action for state (x,y) from q_table.
  #
  # Args:
  #   x, y: state coordinates.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  action_greedy <- which(q_table[x, y, ] == max(q_table[x, y, ]))
  if (length(action_greedy) != 1)
    action_greedy <- sample(x=action_greedy, size=1) # break ties at random

  return (action_greedy)

}

EpsilonGreedyPolicy <- function(x, y, epsilon){

  # Get an epsilon-greedy action for state (x,y) from q_table.
  #
  # Args:
  #   x, y: state coordinates.
  #   epsilon: probability of acting greedily.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  action_max <- which(q_table[x, y, ] == max(q_table[x, y, ]))
  if (length(action_max) != 1)
    action_max <- sample(x=action_max, size=1)  # break ties at random

  # Sample a maximal value action with probability (1-epsilon) or a random action
  # with probability epsilon
  if (sample(x=c(TRUE, FALSE), size=1, prob=c(1-epsilon, epsilon))) {
    action_eps <- action_max
  } else {
    action_eps <- sample(x=c(1:4), size=1)
  }
```

```
  return(action_eps)
}

transition_model <- function(x, y, action, beta){

  # Computes the new state after given action is taken. The agent will follow the action
  # with probability (1-beta) and slip to the right or left with probability beta/2 each.
  #
  # Args:
  #   x, y: state coordinates.
  #   action: which action the agent takes (in {1,2,3,4}).
  #   beta: probability of the agent slipping to the side when trying to move.
  #   H, W (global variables): environment dimensions.
  #
  # Returns:
  #   The new state after the action has been taken.

  delta <- sample(-1:1, size = 1, prob = c(0.5*beta,1-beta,0.5*beta))
  final_action <- ((action + delta + 3) %% 4) + 1
  foo <- c(x,y) + unlist(action_deltas[final_action])
  foo <- pmax(c(1,1),pmin(foo,c(H,W)))

  return (foo)
}

q_learning <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95,
                       beta = 0){

  # Perform one episode of Q-learning. The agent should move around in the
  # environment using the given transition model and update the Q-table.
  # The episode ends when the agent reaches a terminal state.
  #
  # Args:
  #   start_state: array with two entries, describing the starting position of the agent.
  #   epsilon (optional): probability of acting greedily.
  #   alpha (optional): learning rate.
  #   gamma (optional): discount factor.
  #   beta (optional): slipping factor.
  #   reward_map (global variable): a HxW array containing the reward given at each state.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
  #   reward: reward received in the episode.
  #   correction: sum of the temporal difference correction terms over the episode.
  #   q_table (global variable): Recall that R passes arguments by value. So, q_table being
  #   a global variable can be modified with the superassigment operator <<-.

  s <- start_state
  episode_correction <- 0

  repeat{
    # Follow policy, execute action, get reward.
```

```
    # Follow policy, get action
    action <- EpsilonGreedyPolicy(x=s[1], y=s[2], epsilon=epsilon)

    # Take action, get reward
    s_new  <- transition_model(x=s[1], y=s[2], action=action, beta=beta)
    reward <- reward_map[s_new[1], s_new[2]]

    # Q-table update
    correction <- reward +
                  gamma*q_table[s_new[1], s_new[2], GreedyPolicy(s_new[1], s_new[2])] -
                  q_table[s[1], s[2], action]

    q_table[s[1], s[2], action] <<- q_table[s[1], s[2], action] + alpha*correction

    # Episode correction
    episode_correction <- episode_correction + correction
    s <- s_new

    if(reward!=0)

      # End episode.
      return (c(reward,episode_correction))
  }

}
#################################################################
# Q-Learning Environments
#################################################################

# Environment A (learning)

H <- 5
W <- 7

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[3,6] <- 10
reward_map[2:4,3] <- -1

q_table <- array(0,dim = c(H,W,4))

vis_environment()

for(i in 1:10000){
  foo <- q_learning(start_state = c(3,1))

  if(any(i==c(10,100,1000,10000)))
    vis_environment(i)
}

# Environment B (the effect of epsilon and gamma)

H <- 7
W <- 8
```

```r
reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[1,] <- -1
reward_map[7,] <- -1
reward_map[4,5] <- 5
reward_map[4,8] <- 10

q_table <- array(0,dim = c(H,W,4))

vis_environment()

MovingAverage <- function(x, n){

  cx <- c(0,cumsum(x))
  rsum <- (cx[(n+1):length(cx)] - cx[1:(length(cx) - n)]) / n

  return (rsum)
}

for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  for(i in 1:30000){
    foo <- q_learning(gamma = j, start_state = c(4,1))
    reward <- c(reward,foo[1])
    correction <- c(correction,foo[2])
  }

  vis_environment(i, gamma = j)
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
}
for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  for(i in 1:30000){
    foo <- q_learning(epsilon = 0.1, gamma = j, start_state = c(4,1))
    reward <- c(reward,foo[1])
    correction <- c(correction,foo[2])
  }

  vis_environment(i, epsilon = 0.1, gamma = j)
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
}

# Environment C (the effect of beta).

H <- 3
W <- 6
```

```r
reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[1,2:5] <- -1
reward_map[1,6] <- 10

q_table <- array(0,dim = c(H,W,4))

vis_environment()

for(j in c(0,0.2,0.4,0.66)){
  q_table <- array(0,dim = c(H,W,4))

  for(i in 1:10000)
    foo <- q_learning(gamma = 0.6, beta = j, start_state = c(1,1))

  vis_environment(i, gamma = 0.6, beta = j)
}


# By Jose M. Peña and Joel Oskarsson.
# For teaching purposes.
# jose.m.pena@liu.se.

######################################################################
# REINFORCE
######################################################################

# install.packages("keras")
library(keras)

# install.packages("ggplot2")
# install.packages("vctrs")
library(ggplot2)

# If you do not see four arrows in line 19, then do the following:
# File/Reopen with Encoding/UTF-8

arrows <- c("↑", "→", "↓", "←")
action_deltas <- list(c(1,0), # up
                      c(0,1), # right
                      c(-1,0), # down
                      c(0,-1)) # left

vis_prob <- function(goal, episodes = 0){

  # Visualize an environment with rewards.
  # Probabilities for all actions are displayed on the edges of each tile.
  # The (greedy) policy for each state is also displayed.
  #
  # Args:
  #   goal: goal coordinates, array with 2 entries.
  #   episodes, epsilon, alpha, gamma, beta (optional): for the figure title.
  #   H, W (global variables): environment dimensions.
```

```
  df <- expand.grid(x=1:H,y=1:W)
  dist <- array(data = NA, dim = c(H,W,4))
  class <- array(data = NA, dim = c(H,W))
  for(i in 1:H)
    for(j in 1:W){
      dist[i,j,] <- DeepPolicy_dist(i,j,goal[1],goal[2])
      foo <- which(dist[i,j,]==max(dist[i,j,]))
      class[i,j] <- ifelse(length(foo)>1,sample(foo, size = 1),foo)
    }

  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,1]),df$x,df$y)
  df$val1 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,2]),df$x,df$y)
  df$val2 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,3]),df$x,df$y)
  df$val3 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,4]),df$x,df$y)
  df$val4 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,class[x,y]),df$x,df$y)
  df$val5 <- as.vector(arrows[foo])
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),"Goal",NA),df$x,df$y)
  df$val6 <- as.vector(foo)

  print(ggplot(df,aes(x = y,y = x)) +
          geom_tile(fill = 'white', colour = 'black') +
          scale_fill_manual(values = c('green')) +
          geom_tile(aes(fill=val6), show.legend = FALSE, colour = 'black') +
          geom_text(aes(label = val1),size = 4,nudge_y = .35,na.rm = TRUE) +
          geom_text(aes(label = val2),size = 4,nudge_x = .35,na.rm = TRUE) +
          geom_text(aes(label = val3),size = 4,nudge_y = -.35,na.rm = TRUE) +
          geom_text(aes(label = val4),size = 4,nudge_x = -.35,na.rm = TRUE) +
          geom_text(aes(label = val5),size = 10,na.rm = TRUE) +
          geom_text(aes(label = val6),size = 10,na.rm = TRUE) +
          ggtitle(paste("Action probabilities after ",episodes," episodes")) +
          theme(plot.title = element_text(hjust = 0.5)) +
          scale_x_continuous(breaks = c(1:W),labels = c(1:W)) +
          scale_y_continuous(breaks = c(1:H),labels = c(1:H)))

}

transition_model <- function(x, y, action, beta){

  # Computes the new state after given action is taken. The agent will follow the action
  # with probability (1-beta) and slip to the right or left with probability beta/2 each.
  #
  # Args:
  #   x, y: state coordinates.
  #   action: which action the agent takes (in {1,2,3,4}).
  #   beta: probability of the agent slipping to the side when trying to move.
  #   H, W (global variables): environment dimensions.
  #
  # Returns:
  #   The new state after the action has been taken.
```

```
  delta <- sample(-1:1, size = 1, prob = c(0.5*beta,1-beta,0.5*beta))
  final_action <- ((action + delta + 3) %% 4) + 1
  foo <- c(x,y) + unlist(action_deltas[final_action])
  foo <- pmax(c(1,1),pmin(foo,c(H,W)))

  return (foo)
}

DeepPolicy_dist <- function(x, y, goal_x, goal_y){

  # Get distribution over actions for state (x,y) and goal (goal_x,goal_y) from the deep policy.
  #
  # Args:
  #   x, y: state coordinates.
  #   goal_x, goal_y: goal coordinates.
  #   model (global variable): NN encoding the policy.
  #
  # Returns:
  #   A distribution over actions.

  foo <- matrix(data = c(x,y,goal_x,goal_y), nrow = 1)

  # return (predict_proba(model, x = foo))
  return (predict_on_batch(model, x = foo)) # Faster.

}

DeepPolicy <- function(x, y, goal_x, goal_y){

  # Get an action for state (x,y) and goal (goal_x,goal_y) from the deep policy.
  #
  # Args:
  #   x, y: state coordinates.
  #   goal_x, goal_y: goal coordinates.
  #   model (global variable): NN encoding the policy.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  foo <- DeepPolicy_dist(x,y,goal_x,goal_y)

  return (sample(1:4, size = 1, prob = foo))

}

DeepPolicy_train <- function(states, actions, goal, gamma){

  # Train the policy network on a rolled out trajectory.
  #
  # Args:
  #   states: array of states visited throughout the trajectory.
  #   actions: array of actions taken throughout the trajectory.
  #   goal: goal coordinates, array with 2 entries.
```

```
  #   gamma: discount factor.

  # Construct batch for training.
  inputs <- matrix(data = states, ncol = 2, byrow = TRUE)
  inputs <- cbind(inputs,rep(goal[1],nrow(inputs)))
  inputs <- cbind(inputs,rep(goal[2],nrow(inputs)))

  targets <- array(data = actions, dim = nrow(inputs))
  targets <- to_categorical(targets-1, num_classes = 4)

  # Sample weights. Reward of 5 for reaching the goal.
  weights <- array(data = 5*(gamma^(nrow(inputs)-1)), dim = nrow(inputs))

  # Train on batch. Note that this runs a SINGLE gradient update.
  train_on_batch(model, x = inputs, y = targets, sample_weight = weights)

}

reinforce_episode <- function(goal, gamma = 0.95, beta = 0){

  # Rolls out a trajectory in the environment until the goal is reached.
  # Then trains the policy using the collected states, actions and rewards.
  #
  # Args:
  #   goal: goal coordinates, array with 2 entries.
  #   gamma (optional): discount factor.
  #   beta (optional): probability of slipping in the transition model.

  # Randomize starting position.
  cur_pos <- goal
  while(all(cur_pos == goal))
    cur_pos <- c(sample(1:H, size = 1),sample(1:W, size = 1))

  states <- NULL
  actions <- NULL

  steps <- 0 # To avoid getting stuck and/or training on unnecessarily long episodes.
  while(steps < 20){
    steps <- steps+1

    # Follow policy and execute action.
    action <- DeepPolicy(cur_pos[1], cur_pos[2], goal[1], goal[2])
    new_pos <- transition_model(cur_pos[1], cur_pos[2], action, beta)

    # Store states and actions.
    states <- c(states,cur_pos)
    actions <- c(actions,action)
    cur_pos <- new_pos

    if(all(new_pos == goal)){
      # Train network.
      DeepPolicy_train(states,actions,goal,gamma)
      break
```

```
    }
  }

}

#####################################################################
# REINFORCE Environments
#####################################################################

# Environment D (training with random goal positions)

H <- 4
W <- 4

# Define the neural network (two hidden layers of 32 units each).
model <- keras_model_sequential()
model %>%
  layer_dense(units = 32, input_shape = c(4), activation = 'relu') %>%
  layer_dense(units = 32, activation = 'relu') %>%
  layer_dense(units = 4, activation = 'softmax')

compile(model, loss = "categorical_crossentropy", optimizer = optimizer_sgd(lr=0.001))

initial_weights <- get_weights(model)

train_goals <- list(c(4,1), c(4,3), c(3,1), c(3,4), c(2,1), c(2,2), c(1,2), c(1,3))
val_goals <- list(c(4,2), c(4,4), c(3,2), c(3,3), c(2,3), c(2,4), c(1,1), c(1,4))

show_validation <- function(episodes){
  for(goal in val_goals)
    vis_prob(goal, episodes)
}

set_weights(model,initial_weights)
show_validation(0)

for(i in 1:5000){
  if(i%%10==0) cat("episode",i,"\n")
  goal <- sample(train_goals, size = 1)
  reinforce_episode(unlist(goal))
}
show_validation(5000)

# Environment E (training with top row goal positions)

train_goals <- list(c(4,1), c(4,2), c(4,3), c(4,4))
val_goals <- list(c(3,4), c(2,3), c(1,1))

set_weights(model,initial_weights)
show_validation(0)

for(i in 1:5000){
  if(i%%10==0) cat("episode", i,"\n")
```

```
  goal <- sample(train_goals, size = 1)
  reinforce_episode(unlist(goal))
}
show_validation(5000)

knitr::include_graphics('Rplot01.png')
knitr::include_graphics('Rplot02.png')
knitr::include_graphics('Rplot03.png')
knitr::include_graphics('Rplot04.png')
knitr::include_graphics('Rplot05.png')
knitr::include_graphics('Rplot06.png')
knitr::include_graphics('Rplot07.png')
knitr::include_graphics('Rplot08.png')
knitr::include_graphics('Rplot09.png')
knitr::include_graphics('Rplot10.png')
knitr::include_graphics('Rplot11.png')
knitr::include_graphics('Rplot12.png')
knitr::include_graphics('Rplot13.png')
knitr::include_graphics('Rplot14.png')
knitr::include_graphics('Rplot15.png')
knitr::include_graphics('Rplot16.png')
knitr::include_graphics('Rplot17.png')
knitr::include_graphics('Rplot18.png')
knitr::include_graphics('Rplot19.png')
knitr::include_graphics('Rplot20.png')
knitr::include_graphics('Rplot21.png')
knitr::include_graphics('Rplot22.png')
```